PROGRAM PREDICTABILITY AND DATA SECURITY*

Charles G. Moore III and Richard Conway

TR 74-212

September 1974

Department of Computer Science
Cornell University
Ithaca, New York 14850

*The preparation of this paper was supported in part by a grant from the National Science Foundation GJ-41115.
PROGRAM PREDICTABILITY AND DATA SECURITY

Charles G. Moore III and Richard Conway
Department of Computer Science
Cornell University

The fundamental question in the enforcement of security objectives in a computing system is the ability to control the capabilities of a particular process so that it cannot exceed the rights permitted to the user who initiated that process. We are interested in exploring this question under the special restriction that the user is required to encode his program in a "high-level language" -- say a language such as ALGOL or PL/I. We first examine the question in terms of existing languages and implementations of those languages and suggest ways in which enforcement could be enhanced by (1) restricting the user's freedom with the full language, (2) altering the source language, and (3) altering the implementation of the language. This paper is a preliminary report on an inquiry still in progress.

The constraint to the use of a high-level language certainly cannot weaken any protection mechanisms that are provided in either the hardware or the operating system since such mechanisms are concerned with the execution of absolute machine instructions and are unaware of their origin. Whether these instructions were produced directly by a user or indirectly through the translation of a process originally encoded in some high-level language is immaterial. Hence the question is simply whether the additional constraint to a high-level language offers some additional opportunity.

In general the capabilities of interest concern the performance of certain actions on certain objects (read a data object, write a data object, execute a procedure object, etc.) and the ability to cause certain events to occur. One must consider the ability to cause actions rather than the ability to perform actions so that indirect actions are included. That is, the capabilities of a particular process must include what it can achieve indirectly by inducing other processes to act.

At least initially we will focus attention on a general access capability -- given access, the user can read, write, execute, etc. as he chooses. This will serve to illustrate the problems and opportunities of the situation; eventually we will have to refine the analysis to distinguish between different types of action and to include the triggering of events.

We assume that we are given a particular data object called a "security matrix". This is an array M whose rows represent users and whose columns represent data objects. $M(i,j) = 1$ if the user $i$ is allowed access to object $j$; $M(i,j) = 0$ otherwise. We are not concerned in the present discussion with the origin of $M$ -- with the issues of law, ethics and practice that determine whether specific entries should be zero or one -- but

* The preparation of this paper was supported in part by a grant from the National Science Foundation (G7-41115)
only with the problem of enforcing the policies represented by an arbitrary matrix \( M \). \( M \) is itself a data object and authorized access to \( M \) is described in one of the columns of \( M \). (This is presumably a column with few ones since users so privileged could obtain access to any object in the system by appropriately altering their row of \( M \).)

We will also ignore the problem of "masquerading". We will assume that somehow we can correctly identify each user and hence associate him correctly with a particular row in \( M \).

We will use the term "process" to refer to the execution of a procedure \( p \) by a particular user \( u \). There is a one-to-one identification of users with processes and we will often use the terms interchangeably. A "procedure" is an algorithm encoded in some high-level language. A given procedure may be executed by more than one process, and a given process may execute more than one procedure. Each process is identified with an "initial" procedure -- corresponding to the "main" procedure in L1. The execution of a process consists of a single execution of the initial procedure, including whatever other procedures may be called from that initial procedure.

Some users are internal -- in effect, processes to perform some "system function". These processes would tend to be relatively permanent, and to be often invoked by other processes to perform some utility service. Other users are external -- in effect real consumers of the service the system provides. For our purposes the distinction does not seem critical since we are interested in the enforcement of the policies of an arbitrary security matrix. It only suggests that we must accommodate the possibility that one process will "use" another.

We assume that there is some unique identifier accorded to each possible procedure in the system so that we can precisely describe a process by listing, in order of invocation, each of the procedures that it executes. (Even if an individual process involves some form of multi-tasking there is presumably still an order in which procedures are initiated.) A process consists of an identification of its user, and an ordered list of its procedures (which, in general, can only be determined ex post facto):

\[
P_i = (u_i; p_{i0}p_{i1}p_{i2}...)
\]

where \( p_{i0} \) is the "initial" procedure of process \( P_i \).

We will neglect the possibility that an individual user may be responsible for more than one process. We can consider such processes to be initiated by different users -- who happen to have the same privileges. This does not seem to rule out any significant problems or opportunities since eventually we must consider the problem of collusion between two or more processes in general, whether or not the users of those processes happen to be the same individual.

We will think of the objects listed in \( M \) as being in some sense public or shared data. We are not initially concerned...
with objects that are local to a single process and endure only for the life of that process. It is useful to think of the protected objects as being the elements of a data-bank. Suppose that there is some process whose sole function is to manage access to that bank. We will consider all of the objects to be local to and owned by this access process. Although ordinarily the data-bank would be relatively long-lived and large, and hence reside in some secondary storage device, this is not initially essential to our discussion. The central and basic issue is to see whether a process can absolutely control access to those data objects that are local to its initial procedure. The access routine is simply a special case in which the objects are extensive and widely used.

A MINIMAL MODEL FOR DATA ACCESS

We believe that the essential characteristics of the problem are present in the following reduced form:

Suppose that within a single program, say in PL/I, there is an external procedure AM. There are n static data objects declared in AM, which may be identified by the integers \( j = 1, 2, \ldots, n \). One of these objects is an array \( M \) with \( m \) rows and \( n \) columns. Procedure AM has three parameters:

\[
AM: \text{PROCEDURE} (I, J, \text{return});
\]

The function of AM is to return to the calling procedure the value of the \( J \)-th data object if and only if \( M(I,J) = 1 \).

The program also includes \( m \) external procedures \( P_i \), \( i = 1, 2, \ldots, m \). These procedures call upon AM to obtain access to the objects owned by AM:

\[
\text{CALL AM}(I,J,\text{return});
\]

where \( I \) is the index of the calling procedure and \( J \) is the index of the desired object. (Kasquerading in this model would be trivially easy, since the calling procedure would only have to present a false value of the first argument. We ignore the problem and assume that the first argument symbolizes some mechanism by which an unforgeable process identifier is made available to the called procedure.)

Against this context the security questions can be precisely stated as follows:

1) Can \( AM \) be shown to absolutely control access to the \( n \) objects according to the policies described in \( M \)?

2) equivalently, can the capabilities of \( P_i \) with respect to access to the \( n \) objects be completely described (and controlled) by the \( I \)'th row of \( M \)?
There is obviously no question of feasibility. Except for some fussing to comply with PL/I's requirements for type-matching of parameters AM's task is entirely straightforward. However, no matter how straightforward it may be we must be concerned with the "correctness" of AM. This correctness is clearly a necessary condition to the protection of the n objects. We can consider correctness from two points of view. First, consider whether or not the source language algorithm is correct with respect to the function to be performed. Although with techniques available today it is doubtful if the correctness of a program the size and complexity of an operating system, or even a compiler, can be established, it does not seem unreasonable to be able to certify the function of a program such as AM would represent. The second view of correctness concerns the implementation of the language -- whether or not the execution of the compiled code faithfully represents the intent of the algorithm. This turns out to be a very interesting question, involving not only the literal correctness of the compiler but also the completeness of the semantic specification of the language.

All that is at issue here is the scope rules of PL/I. Since these clearly specify that none of the $P_i$ is within the scope of any of the n objects, none of the objects should be accessible from any of the $P_i$ except by explicit call of AM. But that is not necessarily true. Although the scope rules are clear and (presumably) correctly implemented, there are several other lacunae in PL/I that render programs in general unpredictable -- hence one cannot exclude the possibility of access outside of AM.

For example, consider the following segment of one of the $P_i$:

```plaintext
P2: PROCEDURE;
    DECLARE X(I);
    ...
    I = 1000;
    Y = X(I);
    ...
    END P2;
```

What object is accessed by the expression X(I)? This is not specified by the semantics of PL/I and hence is not predictable with respect to the source language specification. An entirely "correct" implementation of PL/I is not obliged to guarantee that X(I) does not access one of the objects local to procedure AM and presumably protected by the scope rules of the language. (To be fair, PL/I does define the meaning of X(I) if the subscriptrange condition is enabled, so it is marginally more complete than other high-level languages. But since the user is allowed to disable the subscriptrange testing the result is effectively the same as languages without any testing.) There are several other similar omissions in the specifications of PL/I that make the actions and accesses of a program unpredictable with respect to the semantics of the language. Only with detailed knowledge of the particular implementation and, in
many cases, of the course of a particular execution, could one predict the action of a given program. This obviously vitiates any hope of controlling the capabilities of a process in a high-level language.

COMPLETENESS OF SEMANTIC SPECIFICATIONS

The remedy is to insist that the semantics of a language be specified "completely" so that the correctness of an implementation would be a sufficient condition to ensure the predictability of a process. One might consider the following as a criterion for completeness:

1. Each operation in the language must be completely defined — that is, a result must be defined for all possible values of its operands. "Error" is, of course, an allowable result but must be well-defined.

2. If an operation is defined only for operands of certain value type, then type-checking must precede each execution of the operation. A type-checking failure is defined to be an error result.

These standards would oblige an implementation to provide sufficient testing (either at translation or execution time) that each operation would be predictable from the semantics of the language. A program, consisting of some sequence of predictable operations, would itself be predictable from the semantics.

However, these standards may be unnecessarily stringent. Consider their application to the binary (two operands) operation of addition. Suppose this were defined as a certain arithmetic operation upon all possible values of its operands — without any provision for overflow or type-checking. Results would at best be addition modulo the maximum number representable in the particular hardware, and at worst be a curious sum of the binary representation of coded values (floating point numbers, character strings, etc.). The point is that while this weak definition of addition probably renders the results of executing a process meaningless to the user, they pose no threat to the predictability of a process from our point of view.

On the other hand the binary operation of subscripting is more sensitive. If a simple algorithm is applied to the two operands, without regard to their value or type, the result is unpredictable from the semantics, and depends upon the relative positioning by the implementation of objects that have no apparent relation to the operands of the subscripting operation.

The distinction seemed to lie in the fact that addition produces a value, whereas subscripting produces a name, which in turn is the argument of an implicit operation called "access". The result of an addition may be meaningless — but it is meaningless to the user — whose actions we are not trying to predict. If subscripting produces a meaningless result the victim is the language processor which will treat that result as if it were a name. Subscripting is, of course, not the
only operation of this character. In PL/I the parameter passing mechanism and the use of pointer variables are similarly sensitive. In all such cases the semantics must define what are meaningful results and the implementation must ensure that this definition is enforced. If a user is offered the opportunity to suspend this definition, presumably in the name of execution efficiency, he also gains the opportunity to contrive a program that can neither be predicted nor controlled based on the semantics of the language.

Another threat to the security of the objects protected by AM will arise if any of the P's could effect a surreptitious entry to AM. If the source language includes label or procedure variables and the user can contrive a branch or call to an object that is meaningless (but not defined as an error) then he can again achieve a program that is not predictable with respect to the semantics of the language. The "transfer function", like the access function, is critical to predictability and sensitive to the value and type of its arguments. It must be protected by a complete definition and a correct implementation.

The sensitive constructs in the source language all seem to have in common the involvement of a syntactic variable whose value is an object (or generally, an object name). When a name is given in the source language as a constant there seems to be little problem. The semantics are explicit and adequately enforced. When the name is given as a variable whose value will become a specific name during execution, the definition is often incomplete, hence an implementation is not bound to provide adequate checking that the values produced are in fact meaningful names.

The completeness (in this sense that we have not yet made precise) of semantic definition, and the correctness of an implementation relative to that definition should ensure a property that might be called the closure of a language. This implies that every program in the language is predictable; none can escape explanation by the semantics. Since every critical operation is completely defined and fully checked before execution there is no sequence of operations that can produce an object other than those for which it was defined.

There are numerous examples of languages (or language subsets) where specific implementations have implied a semantic definition possessing the property of closure. In most cases closure was sought for reasons other than data security, but the examples are nevertheless interesting and suggestive.

PL/C (ref 1) is an implementation of a closed subset of PL/I. All of the critical transfer and access functions are fully checked. The user is prevented from using the
NOSUBSCRIPTRANGE prefix and none of the other checking is accessible from the source language. This extensive checking has two purposes. The first is increased diagnostic assistance to the user. There is, as in PL/I, the implied assumption that the user is not intentionally trying to escape the semantics of the language, so that such events indicate program errors that should be called to his attention. Secondly, and more suggestive in present context, PL/C is designed to process a group of wholly independent user programs in a batch -- that is, to be run as a single job as far as the operating system is concerned. The program mechanism to achieve this batching is co-resident with each user program and must be guaranteed absolute security from actions of the user program if it is to retain the ability to reinitialize the compiler and run the next program of the batch -- regardless of the action and outcome of the last. The only relevant omission in the PL/C subset of PL/I is pointer variables, but it would appear that this feature could be implemented in a manner that would preserve closure.

Another example is a file management system called ASAP (ref 2). This was designed and implemented to explore and illustrate various security features, and closure was deliberately sought in the design.

THE TIMING OF ACCESS CONTROL

Consider further the simplified model of page 3. Procedure AM presumably begins with a routine that checks the appropriate entry in the security matrix to determine whether or not the call is legitimate. This routine would be exercised each time AM was called; even for example, when a particular P_i called AM once for each of the thousands of records in a file. Suppose that there was a secondary entry-point, AM2, following the checking routine. If "trustworthy" processes could be permitted to call at AM2, rather than AM, the burden of repeated access checking could be avoided.

In many cases the complete nature of an access request is specified in terms of constant names in the source program, so that it would be possible to check the permissibility of the access as the program is compiled rather than as it is executed. That is, if the compiler received the identity of the user I as a parameter, and had access during compilation to the security matrix M, it could:

a. compile a call to AM2 if M(I,J) = 1
b. report a "syntax" error (and suppress execution) if M(I,J) = 0
c. compile a call to AM if the specific value of j will not be known until execution time.

The modification of a compiler to provide such checking and hence produce trustworthy programs would be relatively
straightforward. A complication arises in the treatment of the program produced by such a compiler. The program itself becomes an object requiring protection by the system -- particularly from the user who originated it. During the interval between checking access requests (at translation) and executing those requests the program cannot be subject to modification. Moreover, the program is trustworthy only relative to a particular generation of M, hence changes in M would require retranslation of affected programs. The feasibility of such a system has already been demonstrated (ref 2).

PROBLEMS OF INDIRECT ACCESS

Up to this point, we have assumed that the condition
\[ M(I,J) = 0 \] and the "correctness" of AM together imply that process I cannot access object J. This is true only in the strictest sense: if process I "controls" process K, and \[ M(K,J) = 1, \] then process I has an effective capability to access J. Process I, for example, might ask process K to copy the value of the protected object J into some mutually accessible object J' (i.e., some object J' such that \[ M(I,J') = 1 \] and \[ M(K,J') = 1 \]). In fact, two processes can carry on arbitrarily complex communications if there exists even one shared object. (An object is shared if it is accessible to more than one process.) Thus it seems that if processes are not completely isolated from each other (the column sums of M are 1), then we cannot conclude anything about the "indirect" or "effective" capabilities of a process.

The only way out of this dilemma is to make use of information about the actual programs being executed by the processes in the system. If we can examine the program being executed by process I, we might be able to conclude, for example, that the process does not reveal the value of some protected J by placing that value in some shared object. We are talking then about certifying certain properties of programs: properties we might call "security properties". It seems entirely possible to prove some useful security properties of programs written in ALGOL-like languages.

Ordinarily a block-structured languages specifies basic rules for the scope of objects in terms of static structural properties of the program. That is, whether or not a particular procedure can access a particular object depends upon where that procedure was written relative to where the object is owned. The rules are entirely democratic -- any procedure written in the same relative position enjoys precisely the same access. For example, if the structure of a program is:
M: BEGIN;
    (example 1)
    "...
    PL: PROCEDURE;
    X(J) = A(J) + B;
    END PL;
    P2: PROCEDURE
    ...
    END P2;
    END M;

then since PL and P2 are written in the same relative position any object that is accessible to PL is also accessible to P2.

The M matrix of the previous discussion is, in effect, an attempt to discriminate among procedures based on their name, or user identity, in addition to their static location. It extends a facility that is built into the language with one that is programmed using the language. Alternatively one might consider extending the built-in facility. There are various ways that scope might be made explicit by name. For example, a scope attribute could be added to PL/I:

DECLARE X SCOPE(PL, P2);

to specify by name some subset of the normal structural scope.

Another approach to the question might be to analyze actual as opposed to potential scope or usage of objects. For example, in an assignment statement:

X(I) = A(J) + B;        (example 2)

X could be said to "depend" upon I, A, J and B.

Equivalently, we might say the execution of this statement potentially causes information to flow from the objects I, A, J and B to the object X. Given a program, it is possible to construct an information flow graph (ref 3). Nodes in the graph are objects of the program: variables, parameters, procedures. Arrows in the graph indicate that information may flow from one object to another, in the direction of the arc. For example the statement in example 2 would cause arcs to be drawn from the nodes labeled I, A, J, and B to the node labeled X.

Suppose, then, we are interested in whether a process I can access a protected object J. If we know what procedure process I will start executing, we may be able to answer the question. One examines the information flow graph to see if there is a path in the graph from the object J to some object whose scope includes the procedure process I will start executing. If no such path exists, process I cannot access the object J; even with the cooperation of other processes. The converse, however, is not true. The existence of a path does
mean information will necessarily flow across it: it means that information might flow across it.

It is possible to extend the information flow graph algorithm in two ways:

1. To find the access matrix \( M \) that is implicit in a given program, due to the scope rule and program structure.

2. To find conditional access privileges, those that are data dependent in that they are controlled by the value of the data being accessed, or other objects in the environment.

The ability to recognize, and prove the correctness of, data dependent accessing paths is especially interesting. Consider the following procedure:

\[
\text{GETSAL: PROCEDURE(S);}
\]

\[
\text{IF SALARY < 100 THEN RETURN(SALARY);}
\]

\[
\text{END;}
\]

If the variable \( \text{SALARY} \) is inaccessible outside of the procedure \( \text{GETSAL} \), then we can conclude a process may access \( \text{SALARY} \) only if its value is less than 100. This limitation would be revealed in the information flow graph. It seems that the correctness of many data dependent accessing policies could be mechanically verified by constructing the information flow graphs of the programs which implement the policies.

CONCLUSIONS

A high-level language environment would appear to possess some useful characteristics in terms of controlling capabilities but only if that environment is inescapable. If one hopes to interpret and control the capabilities of processes in terms only of a high level description then there must be an absolute guarantee that the process is completely predictable in terms of the semantics of that language. This means that the semantic specifications of the language must be complete with regard to the critical operations in which an object name is generated, and that the implementation must rigorously enforce these complete definitions. Language constructs in which a syntactic variable is evaluated to a name during execution of a process must be monitored during execution. If they are not, presumably in the interests of enhanced efficiency, then the user is offered an opportunity to escape from the high-level language environment and the language becomes simply a convenience to the user but not a limit upon him.

Given that the language environment is inescapable it offers some interesting possibilities. A modern block-structured language offers sufficient independence of procedures that these
could represent the processes of independent users. In effect, the compiler, "main" procedure and various utility procedures would provide the counterpart of an operating system. The normal scope rules of the language, perhaps slightly extended to permit more flexibility, provide a substantial start toward being able to predict and control the capability of each process. An information flow graph of the system could be constructed to verify its security properties. To be sure, confidence in this control relies on the assumption of correctness of the implementation of a significant section of the compiler, but this is several orders of magnitude less demanding than to assume the correctness of a complete operating system.

The principal consequence of the completeness and predictability is to ensure that objects will be accessible only if referenced by symbolic name in the high-level language. This means that capabilities can be assessed and controlled at the source language level rather than in absolute machine instructions. This also suggests that a significant fraction of the monitoring could be advanced to compile-time rather than rely entirely on run-time interpretation. This should permit a substantial improvement in efficiency -- perhaps enough to offset the additional demands of the monitoring required to ensure predictability.

REFERENCES

