POTENTIAL CAPABILITIES IN ALGOL-LIKE PROGRAMS*

Charles G. Moore III.

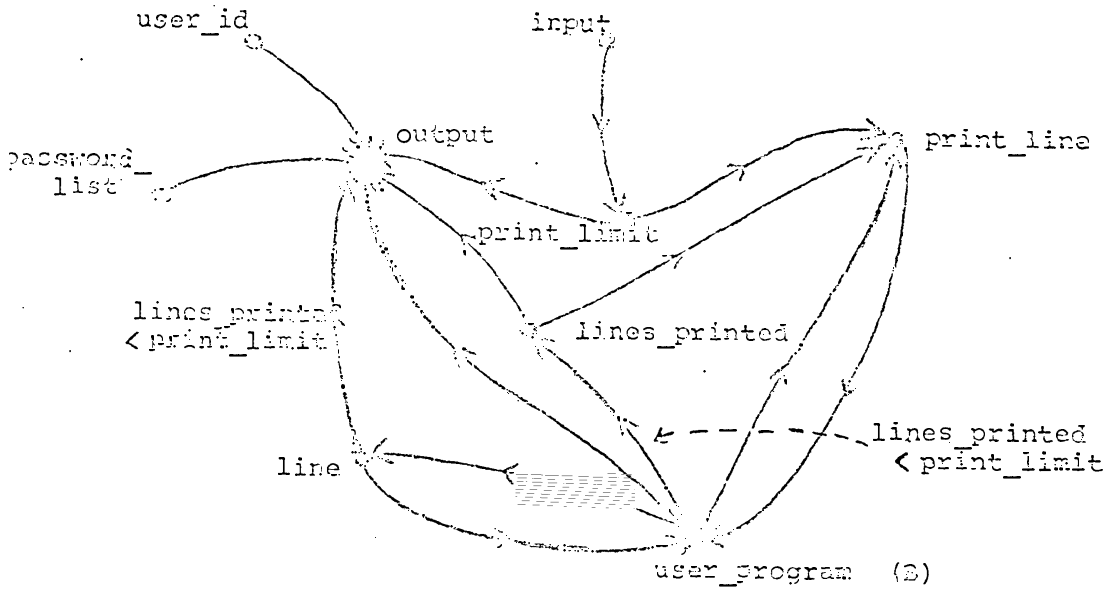
TR 74-211

September 1974

Department of Computer Science
Cornell University
Ithaca, New York 14850

---

*This research was supported in part by NSF grants GJ-28176 and GJ-41115.



We note that the graph shows that B cannot change the value of 'print_limit' (a probably desirable property). The 'password_ list' and 'user_id' cannot be accessed by B. However, information about both these variables escapes through 'output'. Normally, we would want to interpret the ability to output a value (or information about a value), as equivalent to making it accessible to B.

(Some slight simplifications have been made in the above graph. Isolated nodes have been deleted and some conditions simplified. For example, the two arcs from 'input' to 'print_limit' labeled 'id = sys' and '¬id = sys', were reduced to one unconditional arc.)

```
boolean procedure print_line(line); string line;
      begin  if lines_printed < print_limit then
                  begin   output:= line;
                          lines_printed = lines_printed+1;
                          print_line:= true
                  end
             else print_line:= false
      end print_line;

procedure user_program;
          begin B: <stmt>end;  (uninterpreted block)

logon;

end main_program;
```

POTENTIAL CAPABILITIES IN ALGOL-LIKE PROGRAMS*

Charles G. Moore III.

Consider an ALGOL program, with an "uninterpreted block". That is, somewhere in the program text is a "begin...end" block, B, and the statements in that block are unknown. The question we would like to ask (informally) is: What can the statements in B "do", if they are executed? For example, the statements of the block can modify arbitrarily any variable in whose scope they appear. It can call procedures, and through these calls other variables not directly accessible to B (e.g. own variables) may be modified, or their values may become known to the block. In addition, B may be able to directly control the course of future program execution by transferring to some label global to it.

Our question is closely related to a question in the design of operating systems: What can process P do? Operating systems designers have come to talk about this question in terms of capabilities.(1) Capabilities are rights to perform operations on an object (e.g. modify a variable). Systems are designed to prevent untrustworthy programs (more precisely-processes executing untrustworthy programs) from acquiring capabilities whose use might adversely affect the system. The design of such systems is difficult, since capabilities are generally transferred dynamically from process to process during execution. To conclude that a process will not acquire a capability,

---

*This research was supported in part by NSF grants GJ-28176 and GJ-41115.

through any means, is practically impossible in any formal sense, since it requires proofs of properties of programs which pass capabilities (2).

An ALGOL program provides a model for most aspects of this problem in a very direct way. The uninterpreted block B corresponds to the "untrustworthy program" - we don't know what it will do. The set of capabilities possessed by an ALGOL program, at a given point in its execution, is defined by the semantics of the language, as are the ways in which that set changes as execution proceeds. If we are then given the rest of the program text (except for that of the untrustworthy block B), we have some hope of determining what the potential set of capabilities of the block B is.

Our goal in this paper is to present a formal method for answering certain questions about the capabilities of a block B, embedded in an otherwise complete ALGOL-like program. At this point, however, an example will be presented which illustrates informally the concepts and problems involved. The following example represents part of a program in whose security properties we are interested:

(Example 1)

```
string date scope(getdate, setdate);
string procedure getdate; getdate:= date;
procedure setdate(thedate, password);
     if password='QQSV' then date:= thedate;
B: begin...end;
```

In the above example we have extended slightly the syntax of ALGOL to allow explicit scope declarations. The declaration: string date scope(getdate,setdate) is to be interpreted as meaning

the variable date is static (of type own) but may be referenced within the blocks labeled getdate and setdate only. The declaration string declares a variable to be of type character string.

The intent of the example is to implement a variable ("date") which can be read with no restrictions, but can be changed only with an appropriate password ('QQSV'). Clearly, any program may access date by calling the procedure getdate, which returns the value of date as its value. It is equally clear that unless the procedure setdate is called with the string 'QQSV' corresponding to the parameter password, date cannot be modified. However, in the strict sense we must conclude (1) the block B can always change the date and (2) the block B can find out what the password is. This can be done in the obvious way: simply call setdate repeatedly with all possible strings as the password, until the value of date changes. How can this problem be avoided? The answer is to limit the program's freedom to present passwords to setdate, in the following way:

(Example 2)

```
string date scope(getdate, setdate),
string password scope(setpassword, setdate)initial(");

string procedure getdate; getdate:= date;

procedure setdate(thedate); string thedate;
     if password='QQSV' then date:= thedate;

procedure setpassword(the_password); string the_password;
     if password="then date:= thedate;

B:  begin...end
```

It is now impossible for block B to try more than one password, so we can conclude it is unlikely he will be able to change the date, unless he knows the password. It would be incorrect to say he cannot change the date unless he knows the password, because he might guess it. We could guarantee the date could not be changed, from examination of the program text, by deleting the procedure setdate, and the reference to it in the declaration of date.

How then can we prove these statements about a block B? For example, it is "obvious" that date cannot be changed if the setdate procedure is deleted, but not all such properties are so obvious. For example, consider the procedures which implement a stack structure:

(Example 3)

```
integer stack[1:100] scope(push,pop);
integer stacktop scope(push,pop) initial(0);

procedure push(value); integer value;
    if stacktop<100 then
       begin integer i; stacktop:=stacktop+1; i:=value;
             stack(stacktop):= i end;

integer procedure pop;
    if stacktop>0 then
       begin pop=stack(stacktop);
             stacktop:=stacktop-1 end;
```

One security question we might ask about the above example is whether block B can access information placed on the stack and deleted (or "popped off") before B is entered. More precisely, suppose the statements: push(p); q:=pop; have been executed before B is entered. Can B (or statements written in block B) determine the value of p, assuming it is otherwise inaccessible? The answer is yes; block B need consist simply of the statements:

```
B:  begin integer value_of_p;
      push(pop); value_of_p:= pop end;
```

The problem, caused by the call-by-name parameter passing conventions of ALGOL, can be easily fixed. The fact that it did occur, however, should warn us that we need more formal tools for verifying security properties. Of course, the answering of any interesting questions about an arbitrary program will usually be impossible. Someone will quickly show that to answer that question will require, in some cases, that you know whether the program halts (and you don't). It is impossible to avoid this problem entirely. It is possible, however, to give sufficient conditions for program properties to exist; and that will be our approach. We are particularly interested at this point, in one property: the accessibility of a variable V to the block B. We would like to give conditions such that (a) B cannot determine the value of V (to print it out, for example) and (b) B cannot change the value of V. The conditions are related, but not equivalent. Our approach consists of an algorithm which, given a program, constructs a graph. In the graph, nodes generally represent the program variables and arcs represent the flow of information from one variable to another. Once the graph has been constructed the question of whether a variable is accessible is answered by checking the existance of paths in the graph.

Assume we are given an ALGOL program with an uninterpreted begin...end block (which we will call B, whether or not it has that label). Further assume the program has no goto statements, if clauses in expressions, labels, designational expressions or recur-

sive procedure calls. These restrictions on the program are not made to avoid any insoluble problems. They are made to simplify what is intended to be an example of a general method, for use with many programming languages.

The following algorithm constructs a graph for a given program, in which the nodes are variables, formal parameters and procedure names. The arcs represent information flow, and may be labeled. The label, if present, represents a condition on the flow of information (it will not take place unless that condition is satisfied). The program is processed sequentially. Certain actions are taken, depending on the statement type. This will imply recursion for compound statements and blocks. The algorithm has three global data structures, all initially empty.

(1) The information flow graph, which is a directed graph with both nodes and arcs labeled.

(2) A condition list; a list of boolean expressions used to label arcs.

(3) A location list; a list of identifiers. This list contains variables whose value is retained, in some sense, in the locus of control.

The phrase "Process the <stmt>" in the (informal) algorithms which follows means to recursively call the algorithm giving it <stmt> as the <program> to process (the above three data structures remain global to the recursion).*

Let us first define the elements of an expression (either arithmetic or boolean). The elements of an expression will always be some subset of the set of declared variables (<identifier>'s) and is defined as follows:

(1) For each <primary> of the expression which is a <variable identifier>, the <variable identifier> is a member of the set.

(2) For each <primary> of the expression which is a <subscripted variable>, the <array identifier> is a member of the set, and the elements of the subscript expression(s) are members of the set.

(3) For each <primary> of the expression which is a <function designator>, the following algorithm must be performed to determine the <identifier>'s to be added to the set of elements:

(a) For the elements of the expressions which are actual parameters, draw an arcs to the corresponding formal parameter of the procedure (which must already be a node of the graph, see procedure declarations below).

(b) Process the procedure body.

(c) For each node such that there is a path from that node to the procedure name node, add that identifier to the set of elements.

(d) Mark all nodes added in step (b), except those for own variables, and mark all arcs incident on or originating from marked nodes, the formal parameters of the procedure and the procedure name.

(e) For each pair of variables (n,n') such that n and n' are not marked, and for each distinct path of marked arcs from n to n', draw an arc from n to n' and write the conjunction of the conditions on the marked path on the arc.

(f) Delete the marked nodes and arcs.

(Comments: procedure bodies could be 'preprocessed', yielding a great increase in efficiency, but not in clarity. Steps (d),(e),(f) are greatly simplified if procedures are not allowed side effects. Step (e) basically records all the possible side effects.)

We now define the actions to be taken for each type of statement, when processing the program, to construct the information flow graph.

(1) Declaration Statement

(a) If the statement declares variables or arrays, add nodes with these <identifiers> as names to the graph;

(b) If the statement declares a procedure, add nodes for the procedure name and formal parameters to the graph;

(c) Skip the body of the procedure (continue processing with the next statement).

(2) Assignment Statement

(a) Draw arcs from the elements of the right-hand-side expression to the left-hand-side identifier(s).

(b) If a left-hand-side identifier is an <array identifier>, then draw arcs from the elements of the <subscript expression>'(s) to the <array identifer>.

(c) On each arc drawn in (a) and (b), write the conjunction of the expressions on the condition list.

(d) For each <identifier> on the location list, draw arcs to the left-hand-side identifiers.

(3) Conditional Statements [i.e. - if <exp> then <true stmt> else <false stmt>]

(a) Add <exp> to the condition list and all elements of <exp> to the location list;

(b) Process the <true stmt>; replace <exp> with ¬<exp> on the condition list

(c) Process the <false stmt> (if present);

(d) Delete ¬<exp> from the condition list, and all elements of <exp> added at step (a) from the location list.

(4) For statement (for simplicity, we consider only the form: for <variable>:= <exp1> while <exp2> do <stmt>)

(a) Process <variable>:= <exp1> as an assignment statement.

(b) Add <exp2> to the condition list, and the elements of <exp2> to the location list.

(c) Process the <stmt>.

(d) Delete <exp2> from the condition list, and the elements of <exp2> added at step (b) from the location list.

(5) Procedure statements (a statement which is a procedure call)

(a) Execute step (3) of the algorithm to find the elements of an expression. (This step is executed for its side effects).

(6) Compound Statements (begin <stmt> end)

(a) Process the <stmt>'(s)

(7) Uninterpreted block (begin...end)

(a) Add a node (call it B) to the graph to represent B.

(b) Draw arcs from the nodes representing all variables and procedures whose scope includes B, to B.

(c) Draw arcs from B to nodes representing all variables whose scope includes B.

(d) For each procedure whose scope includes B:

(1) Draw arcs to each formal parameter from all potential actual parameters (those nodes referenced in (b), restricted by type correspondence) .

(2) Process the procedure, with B added to the location list.

(Comment: If the uninterpreted block appears in a procedure, the node added at step (a) should not be deleted when the procedure has been processed.)

A number of security properties of an uninterpreted block are now trivially determined. For example:

Access Property: B may determine the value of a variable V only if there is a path from V to B (where B is the node added when processing the uninterpreted block).

Modifying Property: B may modify a variable V (or, more precisely, the value of V may be a function of some action taken by B) only if there exists a path from B to V.

The existence (or non-existance) of the above properties is obviously easy to verify for a given variable. To discover whether B might access the value of some variable V, we construct the information flow graph and see if

there is a path from V to B. Of course the existance of a path does not necessarily mean information will flow along that path. The algorithm "overestimates" the flow of information in that it (1) assumes all possible program paths will be taken, (2) assumes that assignment statements transfer all information about right-hand-side variables to the left-hand-side and (3) assumes tests transfer all information about the elements of the boolean expression to the locus of control. So the access and modify properties are essentially negative in nature: they guarantee certain things cannot happen.

We have not used all the information in the information flow graph. Arcs may be labeled with conditions, and for information to flow along that path, the conjunction of the conditions written on the path must be true. Define the path condition (written $PC(V_1,V_2)$), for some path from a node V_1 to node V_2 as the conjunction of the conditions written on the arcs of the path. Define the flow condition, $FC(V_1,V_2)$, for some node pair (V_1,V_2) to be the disjunction of all path conditions $PC(V_1,V_2)$ for each distinct path from V_1 to V_2. Then we may add to the access property the condition "and FC(V,B) is true"; and to the modify property the condition "and FC(B,V) is true".

It should be made clear that the truth of a flow condition need not be an instantaneous thing: not all terms of some path condition in the flow condition need be simultaneously true. For a flow condition to be true, each term of some path condition must be true at some time. Since flow conditions are arbitrary boolean expressions, whether one will ever be true or not may be difficult to determine (it may even be undecidable).

It is also the case that the access and modify properties are not theorems. Given a formal definition of the semantics of the language, one might be able to prove them. Such an attempt would be interesting. As they stand, however, their correctness depends on the correctness of the algorithm to construct flow graphs. If this algorithm correctly detects all information flow paths, then the properties are correct.

In the appendix, an example of a program and its information flow graph are given.

Implications and Conclusions

The preceding algorithms might well be applied with some success to ALGOL programs, but there is very little practical interest in security properties of ALGOL programs (in fact, there probably isn't any at all). For this approach to secure computing to be practical, several further steps must be made. First of all, the validity of the security properties depends on the information flow graphs, which are constructed according to the semantics of the language. Thus they will be valid only if (1) the semantics of the language are correctly implemented and (2) all programming is done in the language environment.

For (1) to be true, both the compiler for the language, and any "run-time support" must be correct. There is little point in discussing this problem here, not because it is unimportant, but because it is a problem basically unrelated to ours. Certainly the difficulty of the problem will be related to the complexity of the language and the architecture of the actual hardware on which the translated program will run, which are subjects discussed

below in other contexts(4).

The implications of doing all programming in a given language environment (requirement (2) above) are more interesting. In many time-sharing systems, users are required to do all programming in some programming language (e.g. BASIC). Many of the practical security properties of such systems are a result of this constraint. Sitting "behind" the language in such systems, however, is an "operating system", generally written in some other language. If we wish to prove security properties, though, every program must be written in the programming language environment (and therefore in the programming language). Thus we must necessarily view what is usually considered a set of programs with rather arbitrary relationships as a single program. Some of the blocks (or procedures or functions or modules or whatever) may be viewed as operating system components, some as user programs, some as library programs. Their relationship in all cases must be governed by the rules of the language.

This interesting question of what such a language (or language environment) should be like can be considered from two viewpoints. First of all what features would be necessary or desirable from the point of view of programming operating-system-like components? Several things come immediately to mind: some facility for parallel processing; defined data types; more flexible scope and retention rules; pointer type variables. On the other hand, what features make security properties more difficult to verify? Some of the desirable facilities from a programming point of view are disastrous from a security point of view.

Pointer variables are a case in point. Their unrestricted use makes it impossible to conclude what variable (if any) is receiving a value when a pointer variable appears on the left side of an assignment. If their use is restricted, they can become little more awkward to handle (in constructing the information flow graph) than parameters of procedures. We must realize, however, that power and generality in programming features must occassionally be sacrificed to other goals. In this case the "other goal" is the ability to prove security properties of programs.

Appendix

Example of an Information Flow Graph

The following program represents a fragment of a single user operating system. The user is "logged on" by the procedure logon, which checks his user id and password, establishes an output line limit (print_limit), and then calls the user's program (the uninterpreted block), which is the body of the procedure user_program.

We have postulated a new data type device, which allows string input and output through string assignment.

The graph given after the program text is the information flow graph resulting from the application of the construction algorithm to the entire program.

References

(1) Dennis, J.B. and Van Horn, E.C. "Programming Semantics for Multiprogrammed Computation". Journal of the ACM, March 1966, 143-155.

(2) Andrews, G.R., "COPS-A Protection Mechanism for Computer Systems", Ph.D. Dissertation, University of Washington, 1974.

(3) Naur, Peter (et al) "Revised Report on the Algorithmic Language ALGOL 60", Communications of the ACM, January 1963, 1-17.

(4) Organick, E.I., Computer System Organization, Academic Press (New York) 1973.

```
procedure main_program;
device output scope(print_line, logon); device input scope(logon);
integer print_limit scope(logon, print-line);
integer lines_printed scope(print_line) initial(0);
string array user_id[0:n] scope(logon);
string array password_list[0:n] scope(logon);

procedure logon scope(main_program);
      begin string id, password; integer i; boolean success;
           for success:= false while ¬ success do
               begin output:= 'user id:'; id:= input;
                     for i:= 0 while id ¬ = user_id[i] & i ≤ n do
                                i:= i+1;
                     if i < n then success:= true
               end;
           for success:= false while ¬ success do
               begin output:= 'password:'; password:= input;
                     if password = password_list[i] then
                                   success:= true
               end;
           if id = 'sys' then print_limit:= 10000
             else print_limit:= 1000;
           user_program
      end logon;
```