`Proof of Separability´

A Verification Technique for a Class of Security Kernels

(Revised Version of SSM/8)

J.M. Rushby

Computing Laboratory
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
England

Tel.: Newcastle (0632) 329233
Arpanet mail: NUMAC at SRI-CSL

April 20, 1982

ABSTRACT

A formal model of `secure isolation´ between the users of a shared computer system is presented. It is then developed into a security verification technique called `Proof of Separability´ whose basis is to prove that the behaviour perceived by each user of the shared system is indistinguishable from that which could be provided by an unshared machine dedicated to his private use.

Proof of Separability is suitable for the verification of security kernels which enforce the policy of isolation; it explicitly addresses issues relating to the interpretation of instructions and the flow of control (including interrupts) which have been ignored by previous treatments.

-=-=-=-=-=-=-=-=-=-=-

Reprint (slightly expanded) of a paper presented at the
5th International Symposium on Programming,
Turin, Italy, April 4-6 1982.

(Springer-Verlag LNCS No. 137 pp. 352-367)

TABLE OF CONTENTS

`Proof of Separability'

A Verification Technique for a Class of Security Kernels

(Revised Version of SSM/8)

J.M. Rushby

Computing Laboratory
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
England

Tel.: Newcastle (0632) 329233
Arpanet mail: NUMAC at SRI-CSL

INTRODUCTION

Systems with stringent security requirements such as KSOS [Berson79], KVM/370 [Gold79], and the various `Guards' [Hathaway80, Woodward79] are amongst the very first computer systems to be produced under commercial contracts that require formal specification and verification of certain aspects of their behaviour. However, doubts have been expressed concerning whether the techniques used to verify these systems really do provide compelling evidence for their security [Ames79, Rushby81a, Rushby81c]. The purpose of this paper is to develop and justify a new and, it is argued, more appropriate technique for verifying one class of secure systems.

A **secure** system is one which enforces certain restrictions on access to the information which it contains, and on the communication of information between its users. A precise statement of the restrictions to be enforced by a particular system constitutes its **security policy.** Secure systems are needed by the military authorities and by other agencies and institutions which process information of great sensitivity or value. In these environments, it is possible that attempts will be made to gain unauthorized access to valuable information by penetrating the defences of any computer system that processes it. It must be assumed that these attacks may be mounted by skilled and determined users with legitimate access to the system, or even by those involved in its design and implementation. Experience has shown that conventional operating systems cannot withstand this kind of attack, nor can they be modified to do so [Anderson72, Attanasio76, Hebbard80, Linde75, Wilkinson81].

Accordingly, attention has now turned to the construction of **kernelized** systems: the idea is to isolate all (and only) the functions which are essential to the security of the system within a **security kernel.** If the kernel is `correct' in some appropriate sense, then the security of the whole system is assured. The enthusiasm for this

approach is due to the fact that a security kernel can be quite a small component of the total system (it is essentially an operating system nucleus) and there is, in consequence, some hope of getting it right. But since the security of the whole edifice rests on this one component, it is absolutely vital that it is right. Compelling evidence is therefore required to attest to the security provided by a kernel. **Verification**, that is a formal mathematical proof that the kernel conforms to an appropriate and precise specification of `secure behaviour´, is the evidence generally considered to be most convincing [Nibaldi79].

Two main approaches have been proposed for the verification of security kernels. These are **access control verification** [Popek78b] and **information flow analysis** [Feiertag80, Millen76]. The first of these is concerned to prove that all accesses by users to the stored representations of information are in accord with the security policy. For military systems, however, it is not sufficient merely to be sure that users cannot directly access information to which they are not authorized; it is necessary to be sure that information cannot be leaked to unauthorized users by any means whatsoever.

This is the **confinement problem**. It was first identified by Lampson [Lampson73] who enumerated three kinds of channel that can be used to leak information within a system. **Storage channels** are those that exploit system storage such as temporary files and state variables, while **Legitimate channels** involve `piggybacking´ illicit information onto a legal information channel - by modulating message length, for example. The third type of channel is the **covert** one (also called a **timing channel**) which achieves communication by modulating aspects of the system´s performance (for example, its paging rate). Unlike access control verification, information flow analysis can establish the absence of storage and legitimate channels and for this reason it has been the verification technique preferred for certain military systems [Ford78].

Although the properties established by access control verification and by information flow analysis are undoubtedly important ones, it is not clear that they amount to a complete guarantee of security. Both these verification techniques are applied to system descriptions from which certain `low level´ aspects of system behaviour have been abstracted away. Thus autonomous input/output devices - which can modify the system state asynchronously with program execution and which, by raising interrupts, can drastically change the protection state and the sequence of program execution - are absent from the system descriptions whose security is verified by these methods. This is despite the fact that penetration exercises indicate that it is precisely in their handling of these low level details that many computer systems are most vulnerable to attack - and, consequently, that these are the areas where verification of appropriate behaviour is to be most desired.

In a companion paper to this [Rushby81a], I have distinguished between high and low level considerations in secure system design and have proposed that different mechanisms and verification techniques should be employed for each level.

At the high level, the system should be conceived as a distributed one where the significant issues are those of controlling access to information and the communication of information between conceptually

separate single-user machines. The fact that all users actually happen to share the same physical machine should be masked at this level. It is precisely the task of the low level security mechanism to perform this masking and I have proposed a primitive type of kernel called a **separation kernel** to serve this purpose. Its function is to simulate the distributed environment assumed at the higher level of conceptualization. To this end it provides each system component with a **regime** (or `virtual machine´) whose behaviour is indistinguishable from that of a private machine dedicated to that component alone.

The fundamental property to be proved of a separation kernel is that it completely isolates its regimes from one another: there must be absolutely no flow of information from one regime to another. (In practice, controlled flow of information will be required between certain regimes. I will return to this point in the final section of this paper, but for the present I want to concentrate on the simplest case.)

For the reasons outlined earlier, neither of the established methods of security verification is adequate to the task of verifying a separation kernel: we really need a new method. The technique which I propose is a very natural one that is in accord with the intuition underlying the notion of a separation kernel. It is to prove that the system behaves as if it were composed of several totally separate machines - hence the name of this verification technique: `Proof of Separability´.

Although `separability´ is a straightforward notion, its formal definition in terms of a realistic system model is fairly complicated. Possibly, therefore, its own definition may contain errors. The primary purpose of this paper is to convince the reader that this is not the case and that the definition given at the end of this paper is correct. My tactic will be to start off with a specification of `secure isolation´ for a very simple system model and then to elaborate it until I arrive at the definition of Proof of Separability.

THE SPECIFICATION OF SECURE ISOLATION

To begin, we need some formal model of a `computer system´. In developing a verification technique based on information flow analysis, Feiertag and his co-workers used a conventional finite automaton for this purpose [Feiertag77]. At each step, the automaton consumes an input token and changes its internal state in a manner determined by its previous state and by the value of the input token consumed. At the same time, it also emits an output token whose value is determined similarly. Each input and each output token is tagged with its security classification and the specification of security (for the case of isolation) is that the production of outputs of each classification may depend only on the consumption of inputs of that same classification.

While this is an appropriate model for a computer system viewed at a fairly high level of abstraction, it is less realistic as a model for a security kernel. A kernel is essentially an interpreter - it acts as a hardware extension and executes operations on behalf of the regimes which it supports. The identity of the regime on whose behalf it is operating at each instant is not indicated by a tag affixed to the operation by some external agent, but is determined by the kernel´s own state. Furthermore, this model does not capture the instruction

sequencing mechanisms that seem, intuitively, to be of vital importance to the security of a kernel. Accordingly, I shall adopt a slightly different model. I shall suppose that a system comprises a set S of states and progresses from one state to another under its own internal control. The transition from one state to the next will be determined by a NEXTSTATE function solely on the basis of the current state: if s is the current state, then its successor will be NEXTSTATE(s). Naturally, we can think of the value of NEXTSTATE(s) as being the result of `executing' some `operation' selected by a `control mechanism' - although I want to ignore these details for the moment and keep the initial model very austere and general.

Not only must a system have some means of making progress (modelled here by the NEXTSTATE function) but, in order to be interesting, it must interact with its environment in some way: it must consume inputs and produce outputs. For outputs, I shall suppose that certain aspects of the system's internal state are made continually visible to the outside world through a `window' - modelled here by an OUTPUT function: OUTPUT(s) is the visible aspect of state s. Real-world interpretations of this `window' are provided by the device registers of a PDP-11, for example.

In contrast to outputs, which are continuously available, I shall suppose, initially, that inputs are presented to the system just once, right at the start. The INPUT function takes an input value, say i, as its argument and returns a system state as its result. The system, once given this externally determined initial state, thereafter proceeds from state to state under its own internal control. Because INPUT(i) is the initial state of the system, the value of i may be considered to comprise, not merely an input in the conventional sense, but also the `program' which determines the system's subsequent behaviour. The idea that all the input should be presented at one go is clearly artificial since it precludes genuine interaction between the system and its environment. Accordingly, I will extend the model later in order to overcome this objection - but readers should be aware that this extension causes something of a hiatus in the development.

Collecting the burden of the previous discussion together, and proceeding more formally, we may now say that a system or <u>machine</u> M is a 6-tuple

M = (S, I, O, NEXTSTATE, INPUT, OUTPUT)

where:

S is a finite, non-empty set of <u>states</u>,

I is a finite, non-empty set of <u>inputs</u>,

O is a finite, non-empty set of <u>outputs</u>, and

NEXTSTATE: S→S,

INPUT: I→S and

OUTPUT: S→O are total functions. (The reason for assuming <u>total</u> functions will be explained later.)

The computation invoked by an input $i \in I$ is the infinite sequence

$$COMPUTATION(i) = \langle s_0, s_1, s_2, \ldots, s_n, \ldots \rangle$$

where $s_0 = INPUT(i)$ and $s_{j+1} = NEXTSTATE(s_j)$, $\forall\ j \geq 0$.

The result of a computation is simply the sequence of outputs visible to an observer, that is:

$$RESULT(i) = \langle OUTPUT(s_0), OUTPUT(s_1), \ldots, OUTPUT(s_n), \ldots \rangle. \qquad (1)$$

As a notational convenience, I shall allow functions to take sequences as their arguments; the interpretation is that the function is to be applied pointwise to each element of the sequence. Thus, we may rewrite (1) as

$$RESULT(i) = OUTPUT(COMPUTATION(i)).$$

Now, in order to be able to discuss security, we must assume that our machine is shared by more than one user (or else there is no problem to discuss). I shall identify users with the members of a set $C = \{1,2, \ldots, m\}$ of `colours'. Each user must be able to make his own, personal, contribution to the machine's input and be able to observe some part of the output that is private to himself. We can model this formally by supposing that both the sets I and O (of inputs and outputs) are Cartesian products of C-indexed families of sets. That is:

$$I = I^1 \times I^2 \times \ldots \times I^m \quad \text{and} \quad O = O^1 \times O^2 \times \ldots \times O^m.$$

A machine whose input and output sets are of this form will be said to be C-shared. Notice that the components of these products are distinguished by superscripted elements of C. I shall use superscripts consistently for this purpose; subscripts, on the other hand, will be used exclusively to identify the components of sequences.

It is convenient to introduce a projection function, called EXTRACT, to pick out the individual components of members of Cartesian products of C-indexed sets. Thus, when $c \in C$, $i \in I$, and $o \in O$, $EXTRACT(c,i)$ and $EXTRACT(c,o)$ denote the c-coloured components of the input i and the output o, respectively. When a C-shared machine operates on an input $i \in I$, each user sees only his own component of the result. By the convention introduced previously, EXTRACT can be applied to sequences as well as to individuals and so the component of the output visible to user $c \in C$ is the sequence $EXTRACT(c,RESULT(i))$.

Now the simplest and most natural definition of secure isolation is surely that the results seen by each user should depend only on his own contribution to the input. Thus, we require,

$\forall\ c \in C, \forall\ i,j \in I$:

$$EXTRACT(c,i) = EXTRACT(c,j) \Rightarrow EXTRACT(c,RESULT(i)) = EXTRACT(c,RESULT(j)). \qquad (2)$$

Further consideration, however, indicates that this requirement is too strong for our purposes. The real systems, whose salient characteristics we are trying to capture in these definitions, work in a

`time-sharing´ fashion. That is, they first perform a few operations on behalf of one user, then a few more on behalf of another and so on. While the system is operating on behalf of other users, user c should see no change in the outputs visible to him, but the length of time which he must wait before processing resumes on his behalf may well depend upon the activity of those other users. If user c´ can influence the rate at which user c is serviced, and user c can sense his rate of service, then a communication channel exists between c´ and c. This is one of Lampson´s `covert channels´ and while these channels constitute a security flaw and should be countered, their complete exclusion is beyond the scope of simple security kernel technology. Covert channels are typically noisy and of low bandwidth and are normally countered by ad-hoc techniques intended to increase noise and lower bandwidth still further. The threat which I want to completely exclude is that of storage channels and the trouble with (2) as a definition of security is that it requires the absence of covert as well as storage channels. There is little point in demanding the absence of covert channels when this cannot be achieved by the techniques under consideration. We should therefore weaken (2) so that only storage channels are forbidden and this can be done by restricting attention to the sequences of changes in the values of the outputs, rather than the output sequences themselves. To this end, I shall introduce a CONDENSE function on sequences. The CONDENSEd version of a sequence is just the sequence with every subsequence of repeated values replaced by a single copy of that value. For example:

CONDENSE(<1,1,2,2,2,3,4,4,3,3,6,6,6>) = <1,2,3,4,3,6>.

I shall give a precise definition of CONDENSE shortly but first I want to press on with the definition of security. Using CONDENSE, the revised definition of secure isolation, suggested above, becomes:

$\forall$ c $\in$ C, $\forall$ i,j $\in$ I,

EXTRACT(c,i) = EXTRACT(c,j) $\Rightarrow$
CONDENSE(EXTRACT(c,RESULT(i))) = CONDENSE(EXTRACT(c,RESULT(j))).(3)

Unfortunately, this definition is still too strong for our purposes. Suppose that certain inputs cause one of the users to crash the machine, or to loop endlessly when he gets control. This constitutes a type of security breach called `denial of service´. Definition (3) requires the absence of such breaches. Like covert channels, denial of service should be excluded from a secure system - but, again like covert channels, the control of this type of security flaw is beyond the scope of the mechanisms being investigated here, for it concerns the fairness of scheduling procedures and the guaranteed termination of processes. Notice, however, that no attempt to defeat denial of service threats will succeed if any of the basic machine operations can fail to terminate. It is for this reason that I have required all the functions that comprise my machine definitions to be total. Clearly, this requirement must be verified during any practical application of techniques derived from these definitions. Cristian [Cristian81] discusses these issues in a wider context.

We can weaken the definition given by (3) so that it admits denial of service while still excluding storage channels if we require only that the condensed results should be equal as far as they go: we don´t

mind if one of them stops while the other carries on, so long as they are equal while they both survive. I shall, therefore, define a weaker form of equivalence on sequences, denoted by $\cong$, and defined by:

$X \cong Y$ if and only if either $X = Y$ or the shorter of X and Y is an initial subsequence of the other.

My final definition of security is then:

$\forall\, c \in C, \forall\, i,j \in I,$

$$\text{EXTRACT}(c,i) = \text{EXTRACT}(c,j) \Rightarrow \text{CONDENSE}(\text{EXTRACT}(c,\text{RESULT}(i))) \cong \text{CONDENSE}(\text{EXTRACT}(c,\text{RESULT}(j))). \quad (4)$$

Given that we can accept (4) as a precise specification of secure isolation, our task now is to derive a series of testable conditions that are sufficient to ensure this property.

THE VERIFICATION OF SECURE ISOLATION

Before embarking on the main development, we need a precise definition of the CONDENSE function.

Let $X = \langle x_0, x_1, x_2, \ldots \rangle$, be a sequence (either finite or infinite) and let INDICES(X) denote the set of indices appearing in X. Thus

$$\text{INDICES}(X) = \begin{cases} \mathbf{N} \text{ if X is infinite, and} \\ \{0,1,2, \ldots ,n\} \text{ if X is finite and ends with } x_n. \end{cases}$$

Now define the total function $f:\text{INDICES}(X) \rightarrow \mathbf{N}$ by

$f(0) = 0$, and, for $j \geq 0$

$$f(j+1) = \begin{cases} f(j) & \text{if } x_{j+1} = x_j \\ f(j)+1 & \text{otherwise.} \end{cases}$$

$f(j)$ is the number of changes of value in the sequence X, prior to x_j; it is called the condenser function for X.

The range of f is either the whole of $\mathbf{N}$, or else it is a finite set $\{0,1,2, \ldots ,p\}$ for some $p \geq 0$. In either case, let R denote the range of f and let $\bar{f}$ be a right inverse for f. That is, any function such that $f(\bar{f}(j)) = j, \forall j \in R$. (At least one such $\bar{f}$ exists - $\bar{f}(j)$ is the index of an element of X that is preceded by j changes of value.) Then define CONDENSE(X) to be the sequence:

$$\text{CONDENSE}(X) = \langle x_{\bar{f}(0)}, x_{\bar{f}(1)}, \ldots \rangle$$

Notice that this sequence is independent of the choice of $\bar{f}$, for if g and h are any two functions such that

$f(g(j)) = f(h(j)) = j$, then

$$x_{g(j)} = x_{h(j)}$$

even though it is not necessarily true that g(j) = h(j). (Proof by contradiction.)

We shall need the following result concerning condensed sequences:

Lemma 1

Let $X = \langle x_0, x_1, x_2, \ldots \rangle$ and $Y = \langle y_0, y_1, y_2, \ldots \rangle$

be sequences (each either finite or infinite) and let

f: INDICES(X) $\rightarrow$ INDICES(Y)

be a total function such that

$f(0) = 0$ and $\forall\ j \geq 0\ x_j = y_{f(j)}$ and

either $f(j+1) = f(j)$

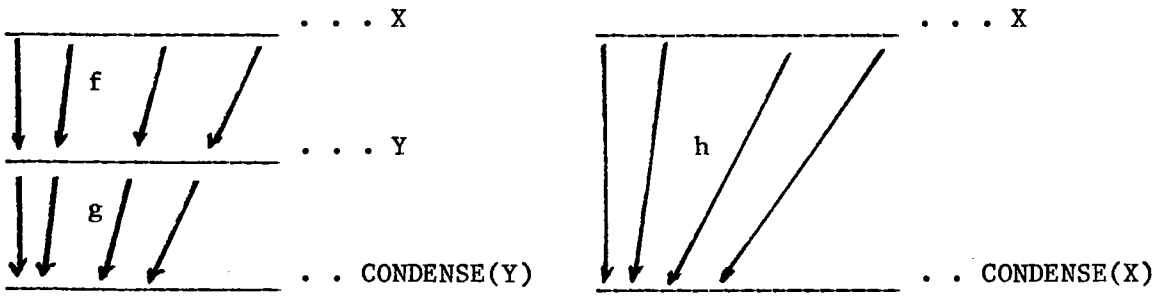
or $f(j+1) = f(j)+1$.

Then CONDENSE(X) $\cong$ CONDENSE(Y).

(Intuitively, this states that if Y is a `slightly' condensed version of X, then X and Y both condense to the same value.)

Proof

Let h and g be condenser functions for X and Y respectively. Pictorially, we have:



and the result is basically a consequence of the fact that h(j) = g(f(j)). Let $\bar{h}, \bar{g}, \bar{f}$ be any three functions such that:

$h(\bar{h}(j)) = j, \forall j$ in the range of h,

$g(\bar{g}(j)) = j, \forall j$ in the range of g,

$f(\bar{f}(j)) = j, \forall j$ in the range of f.

Notice that for any j in the range of h, the value of $x_{\bar{h}(j)}$ is independent of the choice of $\bar{h}$. Similarly, the values of $x_{\bar{f}(1)}$ and $y_{\bar{g}(k)}$ are independent of the precise choice of $\bar{f}$ and $\bar{g}$. Now define

$1(k) = f(\bar{h}(k)), \forall k$ in the range of h.

A straightforward induction on the value of j gives

$$h(j) = g(f(j)), \forall j \in \text{INDICES}(X)$$

and consequently,

$$g(1(k)) = g(f(\bar{h}(k)))$$

$$= h(\bar{h}(k))$$

$$= k. \ (\forall k \text{ in the range of } h).$$

It also follows from the identity $h(j) = g(f(j))$ that the range of h is contained in that of g and hence, for k in the range of h, we have:

$$g(1(k)) = g(\bar{g}(k)) = k.$$

Thus $y_{1(k)} = y_{\bar{g}(k)}$.

But $y_{1(k)} = y_{f(\bar{h}(k))} = x_{\bar{h}(k)}$

and so $x_{\bar{h}(k)} = y_{\bar{g}(k)}$ and the result follows (since these are the k'th elements of CONDENSE(X) and CONDENSE(Y), respectively). □

We now return to the main thread of the argument. Given that we accept (4) as a definition of security, how might we establish the presence of this property? Essentially, (4) stipulates that each user of a C-shared machine must be unaware of the activity, or even the existence, of any other user: it must appear to him that he has the machine to himself. It is a natural and attractive idea, then, to postulate a 'private' machine which he does have to himself and to establish (4) by proving that user c is unable to distinguish the behaviour of the C-shared machine from that which could be provided by a private machine. I shall now make these ideas precise.

Let M = (S, I, O, NEXTSTATE, INPUT, OUTPUT) be C-shared machine where

$$I = I^1 \times I^2 \times \ \ldots \ \times I^n \quad \text{and} \quad O = O^1 \times O^2 \times \ \ldots \ \times O^n,$$

and let $c \in C$. A private machine for $c \in C$ is one with input set I^c and output set O^c, say

$$M^c = (S^c, I^c, O^c, \text{NEXTSTATE}^c, \text{INPUT}^c, \text{OUTPUT}^c).$$

Denote the computation and result functions of M^c by COMPUTATION^c and RESULT^c, respectively. Then M^c is an M-compatible private machine for c if:

$\forall \ i \in I,$

$$\text{CONDENSE}(\text{EXTRACT}(c,\text{RESULT}(i))) \cong \text{CONDENSE}(\text{RESULT}^c(\text{EXTRACT}(c,i))) \qquad (5)$$

That is (roughly speaking), the result obtained when an M-compatible private machine is applied to the c-component of a C-shared input must equal the c-component of the result produced by the C-shared machine M applied to the whole input.

Obviously, we have:

Theorem 1

A C-shared machine M is secure, if for each c ∈ C, there exists an M-compatible private machine for c.

Proof Immediate from (4) and (5). □

Let us now consider how we might prove that a given private machine for c is M-compatible. Direct appeal to the definition (5) is unattractive since this involves a property of (possibly infinite) sequences and will almost certainly require a proof by induction. At the cost of restricting the class of machines that we are willing to consider, we can perform the induction once and for all at a meta-level and provide a more convenient set of properties that are sufficient to ensure M-compatibility.

The restriction on the class of machines considered is a perfectly natural one: I shall consider only those C-shared machines which `time share' their activity between their different users. That is, each operation carried out by the the C-shared machine performs some service for just one user. In particular, it simulates one of the operations of that user's private machine. The identity of the user being serviced at any instant is a function of the current state. Consequently I shall require a function COLOUR which takes a state as argument and returns the identity (colour) of the user being serviced (i.e. the user on whose behalf the next state transition is performed). I shall also require the notion of an `abstraction function' between the states of a C-shared machine and those of a private one.

Theorem 2

Let M = (S, I, O, NEXTSTATE, INPUT, OUTPUT) be a C-shared machine and

COLOUR: $S \rightarrow C$ a total function.

Let $M^C = (S^C, I^C, O^C, NEXTSTATE^C, INPUT^C, OUTPUT^C)$ be a private machine for $c \in C$ and $\phi^C: S \rightarrow S^C$ a total function such that $\forall s \in S, \forall i \in I$:

1) $COLOUR(s) = c \Rightarrow \phi^C(NEXTSTATE(s)) = NEXTSTATE^C(\phi^C(s))$,

2) $COLOUR(s) \neq c \Rightarrow \phi^C(NEXTSTATE(s)) = \phi^C(s)$,

3) $\phi^C(INPUT(i)) = INPUT^C(EXTRACT(c,i))$, and

4) $OUTPUT^C(\phi^C(s)) = EXTRACT(c,OUTPUT(s))$.

Then M^C is M-compatible.

Proof

Denote COMPUTATION(i) by P and $COMPUTATION^C(EXTRACT(c,i))$ by Q where

$P = \langle p_0, p_1, p_2, \ldots \rangle$ and $Q = \langle q_0, q_1, q_2, \ldots \rangle$.

Next, denote EXTRACT(c,RESULT(i)) by X and $RESULT^c$(EXTRACT(c,i)) by Y where

$$X = \langle x_0, x_1, x_2, \ldots \rangle \quad \text{and} \quad Y = \langle y_0, y_1, y_2, \ldots \rangle.$$

By definition, x_j = EXTRACT(c,OUTPUT(p_j)), and

$$y_j = OUTPUT^c(q_j)$$

and we need to prove

$$CONDENSE(X) \cong CONDENSE(Y).$$

Define f: INDICES(P) $\rightarrow$ INDICES(Q) by

$$f(0) = 0, \text{ and } \forall\ j \geq 0$$

$$f(j+1) = \begin{cases} f(j) & \text{if } COLOUR(p_j) \neq c, \\ f(j)+1 & \text{if } COLOUR(p_j) = c. \end{cases}$$

By an elementary induction on j, it follows from parts 1), 2) and 3) of the statement of the theorem that

$$\Phi^c(p_j) = q_{f(j)}$$

and hence, by part 4) of the statement, that

$$OUTPUT^c(q_{f(j)}) = EXTRACT(c,OUTPUT(p_j)).$$

That is

$$y_{f(j)} = x_j.$$

Thus, f (regarded now as a function from INDICES(X) to INDICES(Y)) satisfies the premises of Lemma 1 and so we conclude

$$CONDENSE(X) \cong CONDENSE(Y)$$

and thereby the theorem. □

Let us take stock of our present position. We can prove that a C-shared machine M is secure by demonstrating the existence of an M-compatible private machine for each of its users. How might we demonstrate the existence of such M-compatible private machines? A highly `constructive' approach would be to actually exhibit a private machine for each user and to prove its M-compatibility using Theorem 2. The conditions of Theorem 2 are straightforward and easily checked - it may even be possible to automate much of this checking. On the other hand, the `constructive' aspect of the approach appears rather laborious: the construction of each private machine must be spelled out to the last detail.

A totally different approach would be a `pure' existence proof. We could seek conditions on M which are sufficient to guarantee, à priori, the existence of M-compatible private machines - without ever needing to actually construct these machines at all. The problem here is to find a

suitable set of conditions: conditions which can be easily checked without being overly restrictive on the class of machines that can be admitted. I doubt that these incompatible requirements can be reconciled in any single set of conditions and so conclude that the search for a `pure' existence proof is not worthwhile.

Since both extreme positions (the fully constructive approach and the pure existence proof) have their drawbacks, it may prove fruitful to examine the middle ground. The idea will be to specify just the `operations' of the private machine constructively and to constrain the behaviour of the C-shared machine so that we can guarantee that the construction of the private machine could be completed. To do this, we shall need to elaborate our model once more.

The machines we have considered until now, though very general, are rather unstructured. I now want to constrain them a little by adding more detail to the method by which a machine proceeds from one state to the next. At present, this happens as an indivisible step, modelled by the NEXTSTATE function. In any real machine, the process is more structured than this: first an `operation' is selected by some `control mechanism' and then it is `executed' to yield the next state.

We can model this by supposing the machine M to be equipped with some set OPS of `operations' where each operation is a total function on states. That is

$$OPS \subseteq S \rightarrow S.$$

Next we suppose the existence of a total function:

$$NEXTOP: S \rightarrow OPS$$

which corresponds to the `control mechanism'. In each state s, NEXTOP(s) is the operation which is applied to s to yield the next state. Thus

$$NEXTSTATE(s) = NEXTOP(s)(s).$$

If machines are constrained to have this (more realistic) form, then the set OPS and the function NEXTOP may replace the monolithic NEXTSTATE function in their definition. We then have the following result (which guarantees the existence of a complete private machine, given a specification of only its operations and abstraction functions):

Theorem 3

Let M = (S, I, O, OPS, NEXTOP, INPUT, OUTPUT) be a (new style) C-shared machine and

$$COLOUR: S \rightarrow C \text{ a total function.}$$

Let $c \in C$ and suppose there exist sets

S^c of states, and

$OPS^c \subseteq S^c \rightarrow S^c$ of (total) operations on S^c

together with (total) abstraction functions:

ϕ^c: S $\rightarrow$ S^c and

$ABOP^c$: OPS $\rightarrow$ OPS^c,

which satisfy, $\forall$ c $\in$ C, $\forall$s,s' $\in$ S, $\forall$op $\in$ OPS, $\forall$i,i' $\in$ I:

1) COLOUR(s) = c $\Rightarrow$ $\phi^c(op(s)) = ABOP^c(op)(\phi^c(s))$,

2) COLOUR(s) $\neq$ c $\Rightarrow$ $\phi^c(op(s)) = \phi^c(s)$,

3) EXTRACT(c,i) = EXTRACT(c,i') $\Rightarrow$ $\phi^c(INPUT(i)) = \phi^c(INPUT(i'))$,

4) $\phi^c(s) = \phi^c(s')$ $\Rightarrow$ EXTRACT(c,OUTPUT(s)) = EXTRACT(c,OUTPUT(s')), and

5) COLOUR(s) = COLOUR(s') = c and $\phi^c(s) = \phi^c(s')$ $\Rightarrow$
NEXTOP(s) = NEXTOP(s').

Then there exists an M-compatible private machine for c.

Proof

Define the function $NEXTOP^c$: $S^c \rightarrow OPS^c$ by:

$NEXTOP^c(\phi^c(s)) = ABOP^c(NEXTOP(s))$, $\forall$ s $\in$ S such that COLOUR(s) = c.

Condition 5) ensures that $NEXTOP^c$ is truly a (single-valued) function. If we define NEXTSTATE: S $\rightarrow$ S and $NEXTSTATE^c$: $S^c \rightarrow S^c$ by

NEXTSTATE(s) = NEXTOP(s)(s) $\forall$ s $\in$ S and

$NEXTSTATE^c(t) = NEXTOP^c(t)(t)$ $\forall$ t $\in$ S^c,

then conditions 1) and 2) above ensure these definitions satisfy conditions 1) and 2) of Theorem 2.

Next, define $INPUT^c$: $I^c \rightarrow S^c$ to be any total function which satisfies

$INPUT^c(EXTRACT(c,i)) = \phi^c(INPUT(i))$, $\forall$ i $\in$ I.

Condition 3) above ensures that such a function exists and that it satisfies condition 3) of Theorem 2.

Finally, define $OUTPUT^c$: $S^c \rightarrow O^c$ to be any total function satisfying

$OUTPUT^c(\phi^c(s)) = EXTRACT(c,OUTPUT(s))$, $\forall$ s $\in$ S.

Condition 4) above ensures the existence of such a function and also that it satisfies condition 4) of Theorem 2.

We have now constructed a private machine

$M^c = (S^c, I^c, O^c, NEXTSTATE^c, INPUT^c, OUTPUT^c)$

which satisfies all the conditions of Theorem 2 and so conclude that M^c is M-compatible. □

We now need to make a final adjustment to the model. The present model accepts input only once and we really want something more realistic than this. Real I/O devices do not initialize the system state, they modify it (by loading values into device registers, or by raising interrupts, for example). It is natural, therefore, that the functionality of INPUT should be changed to:

INPUT: $S \times I \rightarrow S$.

We now need to decide when input occurs. On real machines, the state changes caused by I/O devices occur asynchronously, but not concurrently, with the execution of instructions. We could model this by supposing that the INPUT function is applied just prior to the NEXTSTATE function at each step. But with real machines, input does not always occur at every step: whether a device is able to deliver input at some particular instant may depend partly on its own state (whether it has any input available), partly on that of other devices (which may affect whether it can become the `bus master´), and partly on that of the CPU (which may lock out interrupts). We can model this by allowing the machine to make a non-deterministic choice whether or not to apply the INPUT function at each stage. (Actually, this non-determinism does not influence the choice of security conditions given below.) Thus, the machine is now understood to start off in some arbitrary initial state s_0 and to proceed from state to state by:

first) possibly accepting input from its environment, and

second) executing an operation.

That is, if the current value of the input available from the environment is i and the current state of the machine is s, then its next state will be $NEXTOP(\bar{s})(\bar{s})$, where $\bar{s}$ = INPUT(s,i) if the input is accepted, and $\bar{s}$ = s if it is not.

The problem with these changes is that the behaviour of the new model is not a simple variation on that of the old - it really is a new model altogether. For this reason, it is not possible to deduce the conditions that ensure secure behaviour of the new model from those that have gone before; we have to assert them. This is the hiatus in our orderly progress which I hinted at earlier. However, because the new model is similar to its predecessor, and because we have now gained considerable experience in formulating conditions of this sort, I believe that we can be confident of asserting the correct properties.

The conditions that I propose are just those of the statement of Theorem 3, but with its condition 3) replaced by the following pair of conditions which reflect the changed interpretation of the INPUT function (condition 3a is similar to the previous condition 3; condition 3b is new):

3a) $EXTRACT(c,i) = EXTRACT(c,i') \Rightarrow \phi^c(INPUT(s,i)) = \phi^c(INPUT(s,i'))$,

3b) $\phi^c(s) = \phi^c(s') \Rightarrow \phi^c(INPUT(s,i)) = \phi^c(INPUT(s',i))$.

The reader may wonder why I did not use a model with realistic I/O behaviour right from the start. The reason is that I can find no transparently simple specifications of security (corresponding, for example,

to equations (2) to (4)) for such a model. The definition of Proof of Separability would have to be asserted `out of the blue' and the goal of arguing its correctness would have been worse, rather than better, served.

PROOF OF SEPARABILITY

We have now derived the formal statement of the six conditions that constitute the security verificaton technique which I call `Proof of Separability'. Using `RED' as a more vivid name for the quantified colour c, these conditions may be expressed informally as follows:

1) When an operation is executed on behalf of the RED user, the effects which that user perceives must be capable of complete description in terms of the objects known to him.

2) When an operation is executed on behalf of the RED user, other users should perceive no effects at all.

3a) Only RED I/O devices may affect the state perceived by the RED user.

3b) I/O devices must not be able to cause dissimilar behaviour to be exhibited by states which the RED user perceives as identical.

4) RED I/O devices must not be able to perceive differences between states which the RED user perceives as identical.

5) The selection of the next operation to be executed on behalf of the RED user must only depend on the state of his regime.

Interpreted thus, I believe these six conditions have considerable intuitive appeal as a comprehensive statement of what must be proved in order to establish secure isolation between a number of users sharing a single machine. I hope the development that preceded their formulation has convinced the reader that they are the right conditions.

Of course, even the right conditions will be of no practical use if they are so strong that real systems cannot satisfy them. From this point of view, Proof of Separability suffers from a serious drawback: it is specific to the highly restrictive policy of isolation. Most real systems must allow some communication between their users and the aim of security verification is then to prove that communication only takes place in accordance with a stated policy. It is actually rather easy to modify Proof of Separability so that it does permit some forms of inter-user communication: we simply relax its second condition in a controlled manner. For example, if the RED user is to be allowed to communicate information to the BLACK user through use of the WRITE operation, we just delete requirement 2) of Theorem 3 for the single case where COLOUR(s) = RED, c = BLACK, and op = WRITE. Recent work by Goguen and Meseguer [Goguen81], which allows the precise description of a very general class of security policies, may allow this ad-hoc technique to be given a formal basis.

An elementary example of the application of this verification technique (and a comparison with some others) may be found in [Rushby81c]. Present work is aimed at the verification of a complete security kernel

described by Barnes [Barnes80]. The work described here actually grew out of an attempt to formalize the informal arguments used to claim security for this kernel.

REFERENCES

[Ames79] Ames, S.R. Jr., "Security Kernels: Are they the Answer to the Computer Security Problem?", Presented at the 1979 WESCON Professional Program, San Francisco, CA. (September 1979).

[Anderson72] Anderson, J.P., "Computer Security Technology Planning Study", ESD-TR-73-51 (October 1972). (Two volumes).

[Attanasio76] Attanasio, C.R., P.W. Markstein, and R.J. Phillips, "Penetrating an Operating System: a Study of VM/370 Integrity", IBM Systems Journal Vol. **15**(1), pp.102-116 (1976).

[Barnes80] Barnes, D.H., "Computer Security in the RSRE PPSN", Networks '80, pp.605-620, Online Conferences (June 1980).

[Berson79] Berson, T.A. and G.L. Barksdale Jr., "KSOS - Development Methodology for a Secure Operating System", AFIPS Conference Proceedings Vol. **48**, pp.365-371 (1979).

[Cristian81] Cristian, F., "Robust Data Types", Technical Report 170, Computing Laboratory, University of Newcastle upon Tyne, England (1981). (To appear in Acta Informatica).

[Feiertag77] Feiertag, R.J., K.N. Levitt, and L. Robinson, "Proving Multilevel Security of a System Design", Proceedings of the Sixth ACM Symposium on Operating System Principles, pp.57-65 (1977).

[Feiertag80] Feiertag, R.J., "A Technique for Proving Specifications are Multilevel Secure", CSL-109, SRI International, Menlo Park, CA. (January 1980).

[Ford78] "KSOS Verification Plan", WDL-TR7809, Ford Aerospace and Communications Corporation, Palo Alto, CA. (March 1978).

[Goguen82] Goguen, J.A. and J. Meseguer, "Security Policies and Security Models", Proceedings of the 1982 Symposium on Security and Privacy, Oakland, CA., IEEE Computer Society (April 1982). (To appear).

[Gold79] Gold, B.D. et al., "A Security Retrofit of VM/370", AFIPS Conference Proceedings Vol. **48**, pp.335-344 (1979).

[Hathaway80] Hathaway, A., "LSI Guard System Specification (type A)", Draft, Mitre Corporation, Bedford, MA. (July 1980).

[Hebbard80] Hebbard, B. et al., "A Penetration Analysis of the Michigan Terminal System", ACM Operating Systems Review Vol. **14**(1), pp.7-20 (January 1980).

[Lampson73] Lampson, B.W., "A Note on the Confinement Problem", CACM Vol. **16**(10), pp.613-615 (October 1973).

[Linde75] Linde, R.R., "Operating System Penetration", AFIPS Conference Proceedings Vol. **44**, pp.361-368 (1975).

[Millen76] Millen, J.K., "Security Kernel Validation in Practice", CACM Vol. **19**(5), pp.243-250 (May 1976).

[Nibaldi79] Nibaldi, G.H., "Proposed Technical Evaluation Criteria for Trusted Computer Systems", M79-225, Mitre Corporation, Bedford, MA. (1979).

[Popek78] Popek, G.J. and D.A. Farber, "A Model for Verification of Data Security in Operating Systems", CACM Vol. **21**(9), pp.737-749 (September 1978).

[Rushby81a] Rushby, J.M., "The Design and Verification of Secure Systems", Proceedings of the 8th ACM Symposium on Operating System Principles, Asilomar, CA., pp.12-21 (December 1981).

[Rushby81b] Rushby, J.M., "Verification of Secure Systems", Internal Report SSM/9, Computing Laboratory, University of Newcastle upon Tyne, England (August 1981).

[Wilkinson81] Wilkinson, A.L. et al., "A Penetration Study of a Burroughs Large System", ACM Operating Systems Review Vol. **15**(1), pp.14-25 (January 1981).

[Woodward79] Woodward, J.P.L., "Applications for Multilevel Secure Operating Systems", AFIPS Conference Proceedings Vol. **48**, pp.319-328 (1979).