# CONCEPTS AND CONDITIONS FOR CONFINEMENT

Gregory Andrews

TR 75-241

May 1975

Department of Computer Science
Cornell University
Ithaca, New York 14853

# CONCEPTS AND CONDITIONS FOR CONFINEMENT

Gregory Andrews

ABSTRACT

The confinement problem is concerned with preventing a computational service from divulging information entrusted to it. A model of computer protection is presented and used to formally define the problem and its relation to protection mechanisms. Two types of confinement, one concerned with preventing the direct sending of messages and the other with also preventing the use of covert channels, are explored. For both types, conditions sufficient to insure confinement in terms of the capabilities of computations are presented. The conditions make it possible to identify exactly those objects, if any, which can serve as potential channels. Means for plugging the potential channels are also discussed.

## INTRODUCTION

As more and more information has been placed in computer systems, concern for controlling its use has greatly increased. In response, numerous researchers have examined means for protecting stored information and are addressing the problem of designing secure computer systems. This paper examines one specific aspect of computer security, namely the confinement of computations, such as file systems, so that they do not maliciously or accidently divulge information entrusted to them [1,4,9,11]. In particular, we formally define confinement in terms of a model of computer protection and present conditions and techniques sufficient for its solution. Of importance are that these conditions depend only upon the capabilities of computations, not on any knowledge of the code they execute, and that they provide an explicit means to identify any potential channels. As Lampson has observed [9], confinement is made difficult by the variety of non-obvious ways in which information can flow. This paper will show, however, that even "covert channels" can be identified and plugged without unduly restricting computational power or structure.

## A MODEL OF PROTECTION

The function of any protection mechanism is to limit the actions of computations by controlling their access to stored information. It can be abstractly described by four components[1]:

(1) The set of protected objects.

(2) The protection state which defines objects and all authorized actions.

(3) The set of monitors which validate actions by consulting the protection state.

(4) The set of protection primitives which effect changes in the protection state.

---

[1] Numerous examples showing how existing mechanisms can be described by this model are contained in [1].

With respect to protection, three types of objects are of interest: actors, information structures, and memory. Actors, as the name implies, are the objects who take actions; processes and procedures are the two specific types of concern here. Information structures are logical collections of information, analagous to segments. They can be further divided into control structures which contain the capabilities and state of actors and computing structures which contain code and variables. Each information structure maps into a subset of the final class of objects, memory, which contains all addressable storage media.

At every point in time, the protection state contains descriptors for existing objects and capabilities specifying authorized actions. We assume that each object has a unique name and a mapping[2]. For actors (processes and procedures) the mapping points to the control structures implementing the dynamic environment (e.g. procedure argument list or process message queue) and the local environment of the actor. For information structures the mapping identifies the memory objects which contain the information in the structure. For memory objects, the mapping is simply a set of contiguous addresses. The types of objects, their descriptors and their interrelation is depicted in Figure 1. The protection state also contains capabilities where each capability is a pair: (object name, (access attributes, control attributes)) which is associated with an actor and indicates the type of access and control that actor has for the named object. For example, a process might have "Send" access to a message buffer or "Destroy" control of a child process. A procedure might have capabilities to "Call" another procedure, "Execute" its code, and "Read" and "Write" its local variables.

The third protection system component is the enforcement mechanism, a set of monitors in hardware and/or software which validate actions by consulting the protection state. Their functions is to insure that an action, by an actor, is allowed if and only if that actor has appropriate capabilities to take the attempted action on each operand. For example, loading a register should require "Read" access for the information to be loaded and doing input from a file to a buffer should require "Read" access to the file and "Write" access to the buffer.

Some means must exist to change the protection state when new objects are created or access is shared. For this purpose, any protection system contains protection state primitives of two types: those which merely change object descriptors or capabilities, and those which (potentially) pass execution control to another actor and change the actor's dynamic environment. Examples of the former are Create a process, Grant access, and Allocate memory. Examples of the latter are Call a procedure, Send a message, and Load a program status word.

---

[2] In contrast to other models of protection [5,6,7,8,13], we include the object mappings as part of the protection state. As will be shortly shown it is necessary to make assumptions about the integrity of the physical mapping from information to memory if we are to draw conclusions, based on capabilities, about the powers of computations.

For the remainder of the paper, we shall need some basic definitions related to the model. Let the set of protected objects be denoted by O and let $O = A \cup I \cup M$ where A = the set of actors, I = the set of information structures and M = the set of memory objects. A subscript with any of the above symbols refers to a specific member of the set; for example actor $A_i$. Finally, let $Map(O_i)$ refer to the contents of the mapping field in the descriptor of $O_i$ (see Figure 1).

The set of capabilities in the protection state can be naturally subdivided, by actor, into subsets. This leads to the following definition:

Definition: The domain of actor, $A_i$, denoted $D(A_i)$, is the set of capabilities of $A_i$. It is composed of two sets, $D_\ell(A_i)$ and $D_d(A_i)$ containing the capabilities in the local environment, $Map_\ell(A_i)$, and the dynamic environment, $Map_d(A_i)$, respectively.

Intuitively, this means that within an actor's domain is every object for which he has some access or control; it defines the boundaries within which he has been authorized to act[3]. Our purpose is separating the domain into local and dynamic parts is to distinguish between the semi-permanent capabilities of an actor, those which he has whenever he executes, and the dynamic capabilities which he acquires when he is awakened or called by another actor.

A computation executing on a processor always has one active process and one active procedure. We define the computation's environment as the union of the two active domains.

Definition: The environment of an executing process, $A_i$, and an executing procedure, $A_j$, is $D(A_i) \cup D(A_j)$.

Those protection state primitives which change the environment (e.g. Call or Send) we call environment change primitives. Note that their execution can result in an environment containing completely new process and procedure domains (for example when a process is awakened), just a new procedure domain (on a Call) or just a new dynamic procedure domain (on a recursive Call). If actor $A_i$ executes an environment change primitive which cause $D(A_j)$ to become part of the new environment, we say that $A_i$ transfers to $A_j$. To control the use of these primitives, we assume that $A_i$ can transfer to $A_j$ only if there exists a capability ($A_j$, "Transfer") in $D(A_i)$.

We have previously defined a protection state as a set of object descriptors and actor capabilities. In particular, in any computer system there will be an initial set of descriptors and capabilities which exist when the system is loaded. Call this state $S_0$. Then $S_t$ of the same system is defined to be the protection state after t protection state primitives have been executed, without concern for what actor executed them. The protection state

[3] Others have used terms such as "sphere of protection" [5], "ring of protection" [12], and "execution environment" [7] for our concept of domain.

history consequently provides a trace of the protection dynamics of a system as objects are created and destroyed and capabilities are granted and revoked.

DEFINITION OF CONFINEMENT

The confinement problem results when one actor, the customer, makes use of others, the service, but wants assurance that any information passed to the service is not divulged to a third party. In the remainder of this paper, we define and present conditions for the solution of what we call the message confinement and total confinement problems. A service and customer are defined as follows:

Definitions: A service, $\Sigma$, is a set of actors $\{A_1,A_2,\ldots,A_n\}$ with the properties:

(1) There exists a non-empty set of entries, E, containing all $A_i \in \Sigma$ which can be transferred to by $A_j \notin \Sigma$.

(2) From every $A_i \in E$ it is possible to reach every $A_j \in \Sigma$ by executing a series of authorized transfers.

A customer, C, of service $\Sigma$ is any $A_i \notin \Sigma$ who transfers to $A_j \in E$ and passes a set of parameters $P = \{P_1,\ldots,P_m\}$.

In short, a service is a collection of processes and/or procedures which are all (potentially) involved in performing a computation for a customer. It may have many entry points and many possible paths within it, as is usually the case for a file access system. We are interested in those actors who are actively involved with providing a service to C and who consequently have access to the parameters passed by C.

Definition: Given service $\Sigma$ and customer C, $A_i \in \Sigma$ is engaged by C if

(1) C transfers to $A_i$, or

(2) $A_j \in \Sigma$ transfers to $A_i$ and $A_j$ is engaged by C.

Conversely, $A_i \in \Sigma$ is disengaged from C if

(1) $A_i$ transfers to C, or

(2) $A_i$ transfers to $A_j \in \Sigma$ and $A_j$ is engaged by C.

For procedures, engagement works like a push down stack where a procedure becomes engaged (pushed) on entry and disengaged (popped) on return. For processes, the idea is similar but disengagement occurs when a process sends a message to any engaged process, not necessarily the one who engaged him.

A service could possibly be carrying out actions on behalf of more than one customer at a time; this certainly happens with shared system procedures or the processes in a file system. An insecurity exists, however, if any actor in the service has simultaneous access to the parameters of more than one customer, or if the parameters of an engaged actor are accessible to any other. Therefore we require that every service is safely engaged by which we mean the following:

Definition: A service $\Sigma$ is safely engaged if:

(1) no process $A_i \in \Sigma$ is simultaneously engaged by more than one customer;

(2) any procedure $A_i \in \Sigma$ engaged by more than one customer has different dynamic domains when executing on behalf of the different customers; and

(3) no actor $A_j$ has access to $D_d(A_i)$, $i \neq j$, if $A_i \in \Sigma$ is engaged by a customer.

In short, processes work for one customer at a time, shared procedures are pure (reentrant), and parameters are private.

We are concerned with limiting an engaged actor's ability to transmit the values of its parameters, whether the parameter is a constant or a capability.

Definition: The value of parameter $P_k$, denoted $V(P_k)$ is:

(1) any part of the contents of the parameter itself, and

(2) if the parameter is a capability, any information contained in objects accessible using the parameter.

Definition: $A_i$ can transmit a value $V$ to $A_j$ if:

(1) There exists an object $O_k$ such that $A_i$ can take an action which changes the contents of $O_k$ to satisfy property $\Pi$ in state $S_t$ and $A_j$ can take an action which allows him to determine if $O_k$ has property $\Pi$ in state $S_{t'}$, $t' \geq t$; or

(2) $A_i$ can take an action which causes control to transfer to $A_j$.

Examples of direct value transmission are when one actor writes into a storage location and another subsequently reads it or when one actor passes a parameter to another on an environment change. Many covert channels causing a subtle change in the property of an object or a subtle transfer of control also exist; examples are the passage of system time, ratio of IO to computing, paging rate, trapping frequency, or release of a resource. Lampson describes some of the covert techniques in [9]. In all cases, whenever one actor can predictably change the contents of some object, observable by another actor, the object can be used as an information channel. The two means for covert transmission are summarized in Figure 2.

Our goal is to totally confine a service, namely to prevent it from divulging the value of any input parameter unless explicitly authorized by the customer of the service.

<u>Definition</u>: Assume C is a customer of service $\Sigma$ and passes parameters P. $\Sigma$ is <u>totally confined</u> if for every actor $A_i \in \Sigma$, it is necessary to have authorizing capabilities within P in order for $A_i$ to ever transmit a value $V(P_k)$, $P_k \in P$, to any $A_j \notin \Sigma$, $A_j \neq C$.

Before treating the difficult problem of preventing covert transmission, we first focus upon a more restrictive form of confinement, message confinement, which prevents a service from sending messages by writing into objects or passing parameters. This problem is formalized by the following definitions.

<u>Definition</u>: Actor $A_i$ <u>can change</u> the value of $I_k$ only if:

(1) There exists a capability ($I_k$, "Write") in $D(A_i)$; or

(2) There exists a capability ($I_{k'}$, "Write") in $D(A_i)$ and $Map(I_k) \cap Map(I_{k'}) \neq \phi$.

Actor $A_i$ <u>can examine</u> the value of $I_k$ only if:

(1) There exists a capability ($I_k$, "Read") in $D(A_i)$; or

(2) There exists a capability ($I_{k'}$, "Read") in $D(A_i)$ and $Map(I_k) \cap Map(I_{k'}) \neq \phi$.

<u>Definition:</u> Actor $A_i$ <u>can send a message containing value V</u> to actor $A_j$ if:

(1) There exists an information structure $I_k$ such that $A_i$ can change $I_k$ in state $S_t$ and $A_j$ can examine $I_k$ in state $S_{t'}$, $t' \geq t$; or

(2) $A_i$ can transfer to $A_j$ and pass V as a parameter.

In short, $A_i$ can send a message to $A_j$ if they share an information structure, if they have access to information structures which share the same memory or if $A_i$ can invoke $A_j$. These methods are summarized in Figure 2.

<u>Definition</u>: Assume C is a customer of service $\Sigma$ and passes parameters P. $\Sigma$ is <u>message confined</u> if for every actor $A_i \in \Sigma$, it is necessary to have authorizing capabilities within P in order for $A_i$ to ever send a message containing $V(P_k)$, $P_k \in P$, to any $A_j \notin \Sigma$, $A_j \neq C$.

The difference between sending messages and transmitting values is that in the first case the communication must be explicitly authorized by capabilities whereas in the latter, information may be passed directly or as a side effect of an action; for example, execution time could be dependent on an input value and could thus encode that value. Although a confined service will usually be memoryless [6,7], we do not prohibit the retention of values as long as they are never divulged to another on-line computation. For example, a confined service could compute a bill and store it on tape; a subsequent off-line process could then print the bill. We consider a service such as this to be confined because the safety of data on the tape is subject to the security of human procedures, not the integrity of computations.

MESSAGE CONFINEMENT

Conditions for message confinement will be examined first and then applied to solving the total confinement problem. Confinement can be proven by two different methods. First, by knowing the code of a service, one could prove that the service does not ever divulge its input except by using capabilities in its input parameters. For example, a trivial procedure which simply returns the larger of two input parameters and does not invoke any operating system functions might be readily proven to be totally confined. This technique is of limited use though, both because interesting services, such as those in a file system, are extremely complex and because their code often changes and would necessitate a large re-verification for each change. Consequently, we are more interested in certifying that the service does not have sufficient capabilities in its local domain to divulge its input regardless of what it does. This method is valuable because it is invariant to the size, complexity, or dynamics of the code of the service. As will be shown, a combination of these two approaches is required at times.

In order to develop conditions in terms of actor capabilities, we restrict our attention to protection mechanisms which satisfy three basic properties. First, the mechanism is correct in the sense that every action is monitored and the protection state changes only as the result of executing a protection state primitive. Since we include object descriptors within the state, this means that there must be primitives to effect changes in the mapping from information structures to memory; two possible primitives have been described elsewhere [1,2]. Second, no actor can on his own increase his access to or control over any existing object. This assumption holds if an actor can only grant (copy) a subset of the attributes in a capability. Finally, all parameters passed to an actor are revoked on return. Specifically, if $A_i$ transfers to $A_j$ and passed parameters P then whenever $A_j$ transfers back to $A_i$, $A_j$ loses all access to P. In terms of domains, after $A_i$ transfers to $A_j$, $D_d(A_j) = P$ and after $A_j$ returns to $A_i$, $D_d(A_j) = \phi$. This assumption holds for procedures in most systems because Return invariably revokes parameters. For processes, the situation is slightly more complicated but the Sendanswer primitive of the RC4000 system [3] or any similar primitive which revokes access to inter-process messages is satisfactory.

In order to guarantee that a service is message confined, it is sufficient to prove that all messages sent by an engaged actor stay within the service and that no messages are retained if they could be divulged on a subsequent invocation of the service by a different customer; the latter condition means that the service is memoryless.

Message Confinement Property: Assume that C is a customer of service $\Sigma$, that C transfers to an entry E of $\Sigma$ in state $S_{t_1}$ and passes parameters P, and that E transfers back to C in state $S_{t_2}$. Further assume that $\Sigma$ is safely engaged in all states $S_t$, $t_1 \leq t \leq t_2$. Let $A_i \in \Sigma$ by any actor engaged by C in state $S_t$, $t_1 \leq t \leq t_2$. Then $\Sigma$ is message confined if the following is true for every such $A_i$ and $S_t$:

(1) If $(I_k, \text{"Write"}) \in D_\ell(A_i)$ then no actor $A_j$ can examine the value of $I_k$ in any state $S_{t'}$, $t' \geq t$.

(2) If $(A_j, \text{"Transfer"}) \in D_\ell(A_i)$ then $A_j \in \Sigma$.

(3) If $(I_k, \text{"Read"}) \in D_\ell(A_i)$ in state $S_{t_2}$ then no $A_j \in \Sigma$ can change the value of $I_k$ in any state $S_{t'}$, $t_1 \leq t' \leq t_2$.

Proof: We prove that $A_i \in \Sigma$ cannot send a message to $A_j \notin \Sigma$ by using capabilities in $D_\ell$ (the local environment), that $D_d$ does not contain capabilities authorizing message sending except for those in P, and that each engaged $A_i$ is memoryless.

Part 1: To send a message, $A_i$ needs to change the value of some $I_k$ or to pass parameters to some $A_j$ (Definition of send a message). If $A_i \in \Sigma$ is engaged by C, then $A_i$ cannot change the value of $I_k$ if any actor can examine it (condition (1)) and $A_i$ cannot transfer to an actor not in $\Sigma$ (condition (2)); hence $A_i$ cannot send a message to $A_j \notin \Sigma$ using a capability in $D_\ell(A_i)$. If $A_i \in \Sigma$ is never engaged by C then for the same reasons, $A_i$ cannot receive a message from an engaged actor and thus cannot send a message containing $V(P_k)$ to $A_j \notin \Sigma$.

Part 2: By induction on the number of transfers, n, starting with C engaging $E \in \Sigma$ and ending with E disengaging from C, we prove that $D_d(A_i)$, $A_i \in \Sigma$ and $A_i$ engaged by C, never contains capabilities authorizing message sending that are not in P. First, note that $A_i$ cannot have simultaneous access to the parameters of two customers and no other actor can have access to $D_d(A_i)$ ($\Sigma$ is safely engaged).

(a) n = 1. When C transfers to $E \in \Sigma$ and passes parameters P, $D_d(E) = P$ (assumption on parameter passing). Since there are no capabilities in $D_d(E)$ not in P, E cannot send a message except by using parameters P or by returning to C (which is allowed for message confinement).

(b) Assume that no message can be sent in the state existing after n transfers unless it it authorized by capabilities in P. Assume also that $A_i \in \Sigma$ is executing, that $A_i$ transfers to $A_j$ (transfer number n+1), and that $A_i$ passes parameters P'. Since $A_i$ cannot send a message using capabilities in $D_d(A_i)$ and since there are no message sending capabilities in $D_\ell(A_i)$ (by conditions (1) and (2)), P' cannot contain a capability authorizing message sending that was not in $D_d(A_i)$ (assumption on capability granting). Hence, $A_j$ cannot have capabilities in his dynamic environment which enable him to send a message not authorized by the original parameters P.

<u>Part 3</u>: No actor $A_i \in \Sigma$ can retain access to a message containing $V(P_k)$ because:

(a) If $A_i$ is not engaged by C in state $S_t$, $t_1 \le t \le t_2$, then $A_i$ cannot ever be sent $V(P_k)$ (part (1) of this proof).

(b) No $A_i$ engaged by C has any capabilities in $D_d(A_i)$ when he disengages from C (assumption on parameter passing) or has the ability to "Read" any object using a capability in $D_\ell(A_i)$ if that object could contain a message sent in $S_{t'}$, $t_1 \le t' \le t_2$ (condition (3)).

By the three parts, no $A_i \in \Sigma$ can send a message containing $V(P_k)$ to $A_j \notin \Sigma$ unless authorized by capabilities in P; hence $\Sigma$ is message confined.

* * * * *

A few observations about this property are in order. First, the conditions presented only involve the <u>local</u> capabilities of actors so it is possible to prove that a service is message confined without knowing or caring about the way it is coded or what its environment is. Second, message confined services can perform useful functions since writable objects serving as work areas can be passed as parameters; for example a tax service can be passed a capability for a scratch area or a file access procedure can be passed a capability for a buffer. Third, the conditions presented are only sufficient, not necessary, because of our desire to ignore the service's code; they can be made less restrictive in terms of the manipulation of message channels within the service. A very useful relaxation is allowing an actor in the service to trap to an actor outside; in most systems, the trapping actor cannot control the parameters passed and hence cannot pass on values given to him.

The conditions presented could be checked dynamically by a monitor procedure which gets control at the time of every transfer and capability manipulation in states $S_{t_1}$ to $S_{t_2}$. To avoid the obvious overhead of dynamic verification, the following corollary provides a slightly more restrictive set of conditions which can be checked once, possibly by a compiler, either before or when the service is called.

<u>Corollary (Message Confinement)</u>: Given the hypotheses of the Message Confinement Property, $\Sigma$ is message confined if in <u>any</u> state $S_t$, $t \leq t_1$, the following conditions hold:

(1) $(I_k, \text{"Write"}) \not\in D_\ell(A_i)\ \forall A_i \in \Sigma$ and $\forall I_k \in I$.

(2) If $(A_j, \text{"Transfer"}) \in D_\ell(A_i)$, $A_i \in \Sigma$, then $A_j \in \Sigma$.

(3) If $Map(A_i) = I_k$, $A_i \in \Sigma$, then $(I_k, \text{"Write"}) \not\in D(A_j)\ \forall A_j \in A$.

<u>Proof</u>: Condition (3) of the corollary says that in state $S_t$, no actor can change the descriptor of any actor in $\Sigma$ and consequently no changes to the capabilities in $D_\ell(A_i)$, $A_i \in \Sigma$, can be made. By our assumption on capability granting, no actor can increase his access to an object. Consequently, in every state $S_{t'}$, $t' \geq t$, no actor can change the local capabilities of any $A_i \in \Sigma$. Given this fact, the conditions of the Message Confinement Property are satisfied as follows:

(a) By (1) above no $A_i \in \Sigma$ can write into <u>any</u> $I_k$ so trivially, "If $A_i$ can write $I_k$ then no $A_j$ can examine $I_k$," is true for all states $S_{t'}$, $t' \geq t$.

(b) Condition (2) above is identical to condition (2) of the Message Confinement Property.

(c) Since no $A_i \in \Sigma$ can change the value of any $I_k$ (1 above) trivially "If $A_i$ can read $I_k$ then no $A_j \in \Sigma$ can change $I_k$" is true for all states $S_{t'}$, $t' > t$.

* * * * *

Many useful services in an operating system are in practice message confined. For example, a file access procedure which acquires capabilities for both a memory buffer and an external file (information structure) as input parameters can be easily designed to meet the above conditions. Such a procedure can have read-only access to the file's descriptor or perform security transformations (e.g. encryption or decryption) and still be message confined. In the ideal situation, which exists for simple services, we can apply the above Corollary directly and establish message confinement by examining a service's capabilities. Even for more complex services, such as those which maintain tables of information about each customer's parameters, we can still use the above property to aid us. Let any object which leads to a violation of one of the conditions of the Property or Corollary be called a <u>potential message channel</u>. A service is still message confined if we can prove (by examining the code of the service) that no potential message channel is used to send value $V(P_k)$ for $P_k \in P$ (input parameters) to any actor outside the service. For example, a service may have write access to a shared table (such as for paging) as long as values written into the table are independent of input. Or a file directory service may keep local storage for each customer as long as the customer is correctly identified and only his storage is used when he engages the service. A technique which constructs an information flow graph from a program and determines sufficient conditions to insure that the value of one object is not dependent on another has been developed [10] and could be used to certify potential message channels.

TOTAL CONFINEMENT

Recall that a service is totally confined if no actor in the service can transmit an input value by any means whatsoever to an actor outside the service, unless authorized by the customer. Plugging all message channels is obviously one prerequisite; the other is plugging all covert channels. Sending a message involves changing the value of an object as a direct effect of an action, such as writing into a table or calling a procedure and passing parameters. By contrast, covert transmission involves changing the value of an object as a <u>side effect</u> of an action; if the presence or absence of the side effect can be determined, some information can be transmitted. The following are indicative examples of common actions and possible side effects:

(1) computing anything which advances the system clock;

(2) receiving a message (doing a P on a semaphore) which causes another process to block if it later attempts to receive the same message;

(3) generating a page fault which causes a memory mapping change and the initiation of IO;

(4) accessing a page which may cause a reference or modify bit to change.

A service is totally confined if it is message confined and if no side effects of actions within the service are observable outside. After presenting formal conditions for total confinement, we describe ways in which side effects can be identified and plugged to prevent their use as information channels.

Definition: If $A_i$ executes an authorized action $\alpha$ on operands $O'$, $O' \subseteq O$ and the content of $O_j$ changes during the execution of $\alpha$, $O_j \not\in O'$, then $O_j$ is a side effect of $\alpha$.

Definition: If there exists any authorized action $\alpha$ by which $A_i$ can ascertain knowledge about the contents of $O_j$, then $O_j$ is an information channel to $A_i$.

Total Confinement Property: Given the hypotheses of the Message Confinement Property, $\Sigma$ is totally confined if:

(1) $\Sigma$ is message confined, and

(2) For every actor $A_i \in \Sigma$ engaged by C in state $S_t$:

(a) If $O_k$ is a side effect of action $\alpha$ authorized by capabilities in $D_\ell(A_i)$ then $O_k$ is not an information channel to $A_j \not\in \Sigma$ in state $S_{t'}$, $t' \geq t$.

(b) If by an action of $A_i$, authorized by capabilities in $D_\ell(A_i)$, control transfers to $A_j$ then $A_j \in \Sigma$.

(c) If $O_k$ is an information channel to $A_i \in \Sigma$ in state $S_{t_2}$ then $O_k$ was not a side effect of any authorized action $\alpha$ by $A_j \in \Sigma$ in state $S_{t'}$, $t_1 \leq t' \leq t_2$.

Proof: (Outline) Since $\Sigma$ is message confined (condition (1)) we only need to worry about transmission by covert channels. By arguments parallel to those in the proof of the Message Confinement Property we outline arguments to prove that no side effects of actions taken by $A_i \in \Sigma$ are observable by $A_j \not\in \Sigma$, $A_j \neq C$.

<u>Part 1</u>: Value transmission requires changing some object or transferring control. Conditions 2(a) and 2(b) prevent $A_i \in \Sigma$ from transmitting values to $A_j \notin \Sigma$ during state $S_t$ by using capabilities in $D_\ell(A_i)$.

<u>Part 2</u>: By induction on the number of transfers starting with C engaging $E \in \Sigma$ and ending with E disengaging from C we can prove that an engaged actor $A_i$ has no capabilities in $D_d(A_i)$ authorizing value transmission that are not in P.

<u>Part 3</u>: Each $A_i \in \Sigma$ engaged by C is memoryless because $A_i$ has no access to information channels which could have been used directly (message) or indirectly (side effect) by any $A_j$ engaged by C.

* * * * *

The general observations made about the Message Confinement Property also apply here. Namely, the conditions only involve capabilities, they are sufficient but not necessary, and they are verifiable if side effect objects can be identified. A corollary analagous to Corollary 1 for Message Confinement can also be stated thereby allowing the conditions to be checked statically before the service is ever used.

Once a totally confined service commences execution, everything, including control, must stay within it. If one process causes another to awaken, the second must be within the service or the occurrence of the wakeup could be used to signal 1 bit of information. The only exception is if an actor in $\Sigma$ is pre-empted by an actor outside $\Sigma$, as might happen on an interrupt. As long as the interrupt could not be triggered from within the service, its occurrence could not signal information to any potential listener.

In order to apply the above property, we must be able to identify possible side effects and prevent their use as covert channels. We feel that the possible side effects can be explicitly enumerated for a given system by examining all of the operations available to actors. First, the number of different machine operations is small and the number of side effects at the machine level is consequently small. Second, the primitive (indivisible software) operations of an operating system kernel are also typically small in number and each is small in size. As a result, current proof techniques can be used to ascertain their effects and verify their correctness. In short, for a given machine and nucleus, all side effects should be identifiable.

Even if side effects exist, they cannot be used as covert channels if we can prevent their use from being observed. In most of the examples which have been catalogued, either a clock in conjunction with something else or descriptors for memory or actors is the means for covert transmission. In both cases, access to the potential channel can be denied to a service without causing any apparent restrictions to the service's power. The clock can be made unreadable except by a few trustworthy processes who share no other side effect with the service and access to process descriptors could simply be inaccessible outside the nucleus.

As was the case with message confinement, ideally we can apply the Total Confinement Property directly by looking at an actor's local capabilities, ascertaining thereby the actions he could possibly execute, and then proving that either no side effects exist or that they are not observable to other actors. However, in many cases a potential covert channel exists, for example the locks which prevent simultaneous reading and writing of a file. If we define a potential channel as any side effect object which is observable by others (in violation of one of the conditions of the Property) then a service is totally confined if we can prove that the potential channel is not used by the service in a way dependent on the value of any input parameter. For example, semaphores used for file locks may always be decremented before an attempted file access irregardless of the values in the file or buffer to be transmitted[4]. The existence of the semaphore provides a potential channel but the above usage does not involve an attempt to send information. Again the information flow graphs of [10] give rise to sufficient conditions to insure that a channel is not used covertly.

## CONCLUSIONS

In this paper, we have attempted to characterize the confinement problem in a manner which presents usable techniques for designing and verifying confined services. In particular, we have defined two types of confinement, message and total, in order to separate the hard problems from the (relatively) easier ones. The conditions presented in both Properties are useful in that they provide a direct means, in terms of capabilities, to identify exactly those potential channels which could be used by a service to leak information. Having identified such channels all others can be ignored and attention can be focussed entirely upon the problem of proving that the channel is not used or that its use is independent of parameters of the service. An important point is that if two services are each confined and operations are then added so that on uses another, the resulting service is also confined. This means that levels of abstract machines where each level uses previous ones and implements new operations can potentially be combined into large, confined services in a constructive manner. Preliminary examination of the problem of composing a file system in this manner and using the techniques and theory presented in this paper looks promising. Much work remains to be done, however, to work out the detail of applications, develop algorithms for checking conditions, and relate this theory to the operations of existing and potential systems.

The author gratefully acknowledges the assistance of A. C. Shaw who helped clarify many of the above concepts.

[4] The absence or presence of blocking on the part of the decrementing actor must also be unobservable outside the service.

BIBLIOGRAPHY

1. Andrews, G.R. COPS - A protection mechanism for computer systems. Tech. Report 74-07-12 (Ph.D. Thesis), Computer Science Group, Univ. of Washington, July 1974.

2. Andrews, G.R. COPS - A mechanism for computer protection. Proc. Int. Workshop on Protection, IRIA, Paris, France, August 1974, 5-25.

3. Brinch Hansen, P. The nucleus of a multiprogramming system. Comm. ACM 13, 4(April 1970), 238-250.

4. Denning, D.E., Denning, P.J., and Graham, G.S. Selectively confined subsystems. Proc. Int. Workshop on Protection, IRIA, Paris, France, August 1974, 55-61.

5. Dennis, J.B. and Van Horn, E.C. Programming semantics for multiprogrammed computations, Comm. ACM 9, 3 (March 1966), 143-155.

6. Graham, G.S. and Denning, P.J. Protection-principles and practice. AFIPS Conf. Proc. 40 (1972 SJCC), 417-429.

7. Jones, A.K. Protection in programmed systems. Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, June 1973.

8. Lampson, B.W. Protection. Proc. Fifth Princeton Conf. on Info. Sciences and Systems (March 1971), 437-443. Reprinted in Operating Systems Review 8, 1 (Jan. 1974), 18-24.

9. Lampson, B.W. A note on the confinement problem. Comm. ACM 16, 10(Oct. 1973), 613-615.

10. Moore, C.G. Potential capabilities in ALGOL-like programs. Tech. Report 74-211 (Sept. 1974), Dept. of Computer Science, Cornell University.

11. Popek, G.J. and Kline, C.S. Verifiable secure operating system software. AFIPS Conf. Proc. 43 (1974 NCC), 145-151.

12. Schroeder, M.D. and Saltzer, J.H. A hardware architecture for implementing protection rings. Comm. ACM 15, 3(March 1972), 157-170.

13. Spier, M.J. A model implementation for protective domains. Int. Journal of Computer and Info. Sciences 2, 3(Sept. 1973), 201-229.

Figure 1

Protection Objects and Relations

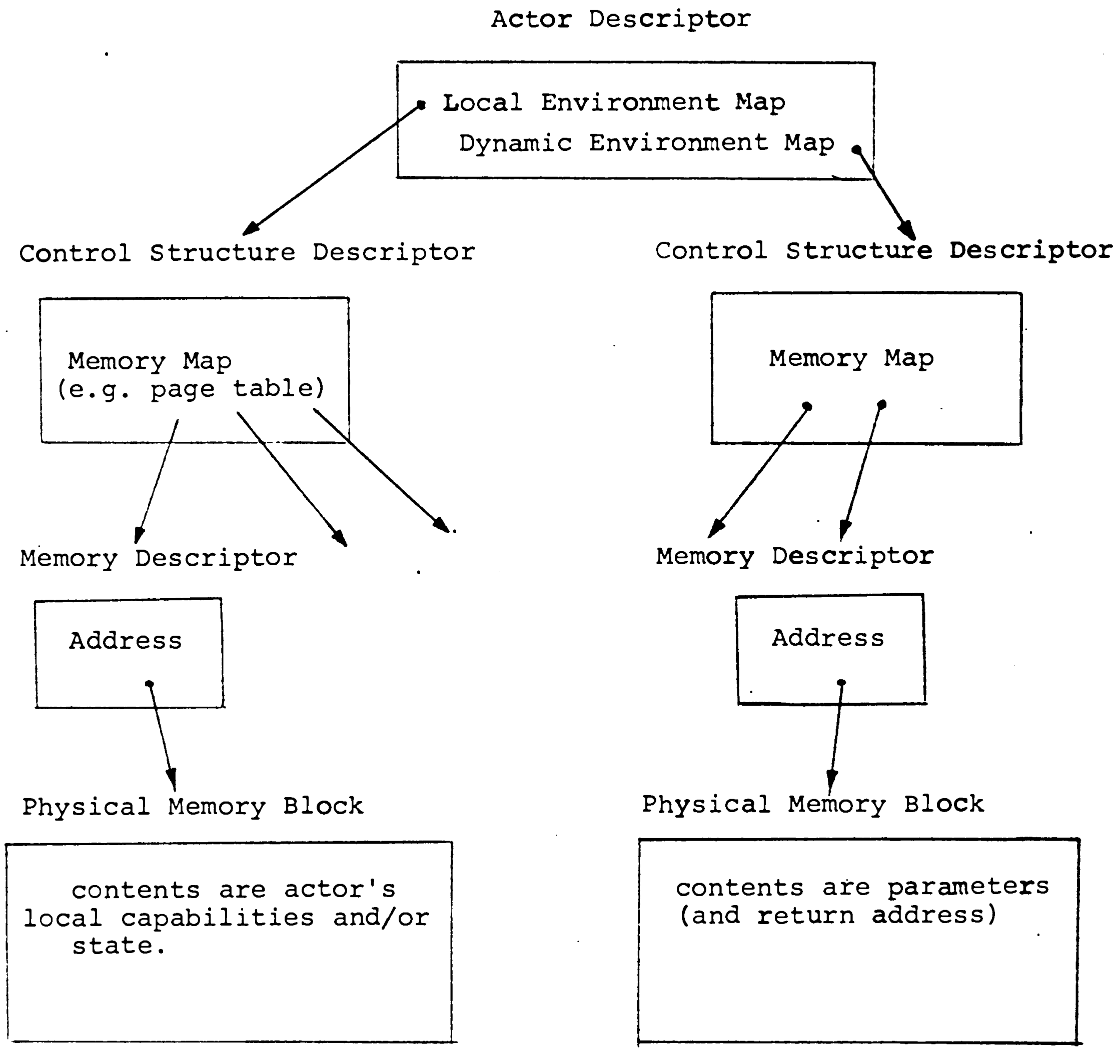
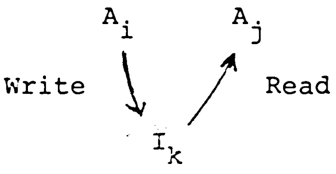


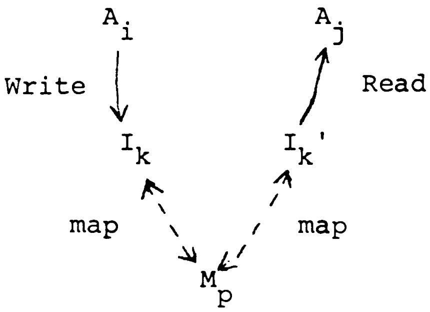
Figure 2

Sending Messages

(a)



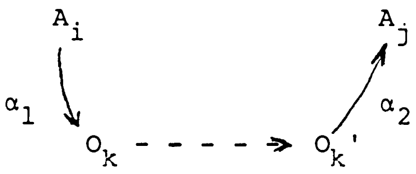

(b)




(c) $A_i \xrightarrow[\text{with parameter}]{\text{Transfer}} A_j$

Covert Transmission of Values

(a)




$\alpha_1$ - action on $O_k$ which cause $O_k$' to have property $\Pi$

$\alpha_2$ - action to determine if $O_k$' has property $\Pi$

(b) $A_i \overset{\alpha}{\dashrightarrow} A_j$

$\alpha$ - action (call, trap, interrupt, release of resource, etc.) by $A_i$ which causes transfer of control to $A_j$