A Practical Executive for Secure Communications*

Gary Grossman
Digital Technology Incorporated
Champaign, Illinois

1. Introduction

Computer communication is now used in many endeavors in which security and privacy are important, both in government and in the private sector. To support the need for secure computer communication, Digital Technology Incorporated (DTI) has developed the Secure HUB** Executive (HUB), a verified secure operating system oriented toward supporting communications and other real-time applications. The Secure HUB Executive currently runs on Digital Equipment Corporation PDP-11 and VAX-11 hardware, but it is portable to a wide range of mini- and microcomputers.

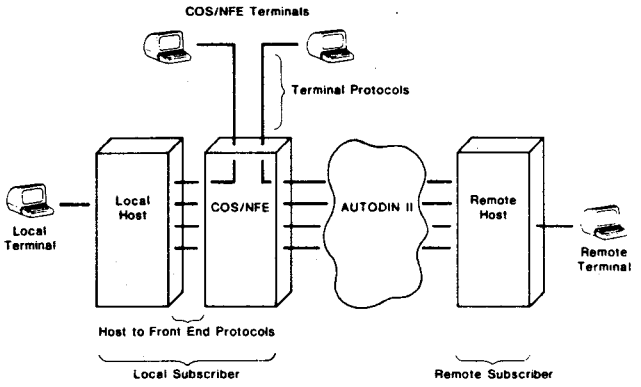
Performance in a communications processor is heavily dependent on the speed at which the operating system can perform interprocess communication and process switching functions. The HUB performs interprocess communications functions approximately two times faster, and process switching functions approximately four times faster, than a general purpose operating system using the same hardware.

The first application of the HUB is in the Communications Operating System Network Front End (COS/NFE)[1] under development at DTI, under contract to the Defense Communications Agency. The COS/NFE is a prototype verifiably secure network front end for the AUTODIN II secure communications network[2].

Other envisioned applications for the HUB include secure gateways between networks at different security levels, secure packet-switching nodes, and secure data acquisition installations.

1.1. Background

The HUB is the operating system that supports the COS/NFE. The COS/NFE is a prototype verifiably multi-level secure communications processor that interfaces a Honeywell H6000 computer to AUTODIN II. The hardware base for the COS/NFE is a Digital Equipment Corporation PDP-11/70 minicomputer. The COS/NFE performs AUTODIN II protocol functions for the H6000. It also interfaces terminals to both the H6000 and AUTODIN II (see Figure 1).



COS/NFE Functions
Figure 1

Except for security, the COS/NFE performs functions identical to those of the Interim Network Front End (INFE)[3]. The INFE is also based on a PDP-11/70, but its operating system, called INFE UNIX***, is a modified version of the Bell Laboratories UNIX operating system[4].

UNIX is a general purpose time sharing operating system that is available on a variety of mini- and microcomputers. The standard UNIX interprocess communication (IPC) mechanisms are inadequate to support high-bandwidth communications. They do not permit the kind of interprocess connectivity that is required, and they introduce a great deal of CPU overhead. UNIX was modified for the INFE to add a new IPC mechanism, called Attach I/O[5]. Attach I/O was designed to provide higher performance than the existing UNIX IPC mechanisms. Even with Attach I/O, 70% of all CPU cycles in the INFE are consumed by IPC and process switching overhead.

The COS/NFE performs the same functions using the same hardware base as the INFE. It thus provides an opportunity for comparing the performance of non-secure INFE UNIX with the performance of the Secure HUB Executive.

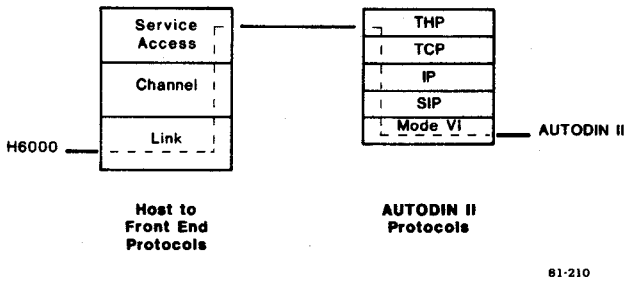
1.2. Protocol processing

In both the INFE and the COS/NFE, several protocols must be interpreted for each message as it travels through the system. For instance, on a communications connection from the H6000, through the COS/NFE or the INFE, to AUTODIN II, eight different protocols may have to be interpreted for each message (see Figure 2). Of these protocols, three are Host to Front End Protocols[6] and five are AUTODIN II protocols.

* The specification and verification work described in this paper were performed under contract DCA-100-79-C-0035 with the Defense Communications Agency.

** HUB is a trademark of Digital Technology Incorporated.

*** UNIX is a trademark of Bell Laboratories.



Network Front End Protocols

Figure 2

In the INFE, each of these protocols is implemented in a separate process (see Figure 3). (Actually, the mapping of protocols to processes in the INFE is slightly different than is shown here, but this is irrelevant for the purposes of this discussion.) This is to maintain functional isolation between the protocol implementations and to ensure that each process will fit into the restricted PDP-11/70 address space (64KB each for program and data). Two of the protocols, the AUTODIN II Transmission Control Protocol (TCP) and the Host to Front End Channel Protocol, perform multiplexing and demultiplexing of connections. As the figure shows, whether a protocol interpreter process handles the connections as a single multiplexed stream or as separate streams, the same process handles all connections that require a particular protocol.

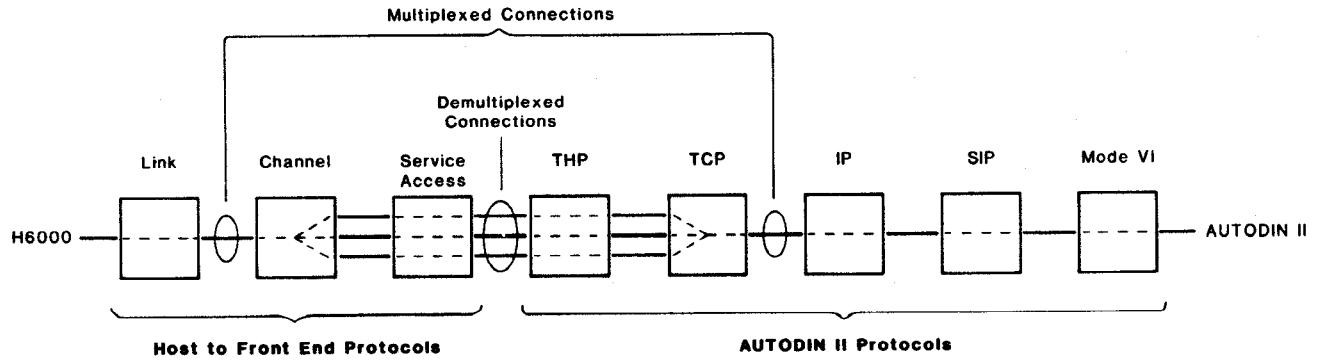
In a system that is to be verified as multi-level secure, handling all connections by the same process, regardless of the security levels of the connections, would require all application level software to be verified. At the current state of the art in software verification, this would be prohibitively expensive, if not infeasible. The amount of software that is to be verified must be reduced to a minimum if verification is to be practical.

To minimize the amount of COS/NFE software that must be verified, software that performs security-related functions is separated from non-security-relevant software. For example, the portions of the Transmission Control Protocol (TCP) implementation that need to be trusted to handle data a multiple security levels will be separated from the portions that do not need to be trusted. The TCP performs functions roughly equivalent to those of the Transport Layer of the International Standards Organization Reference Model for Open Systems Interconnection[7], but it is designed to operate in environments where security and high reliability are required, even when the available transmission media are unreliable.

The TCP has the capability of tagging each message with the security level of its contents. In the COS/NFE multi-level secure environment, this facility of the TCP will be used to demultiplex messages arriving from the multi-level AUTODIN II network into the appropriate separate single security levels. This security-relevant function will be performed by a trusted process, called the TCP Multiplexor, with verified software. Other portions of the TCP, that are not security relevant, and therefore do not have to be trusted, will be implemented in separate processes, called TCP connection machines, with at least one per security level (see Figure 4).

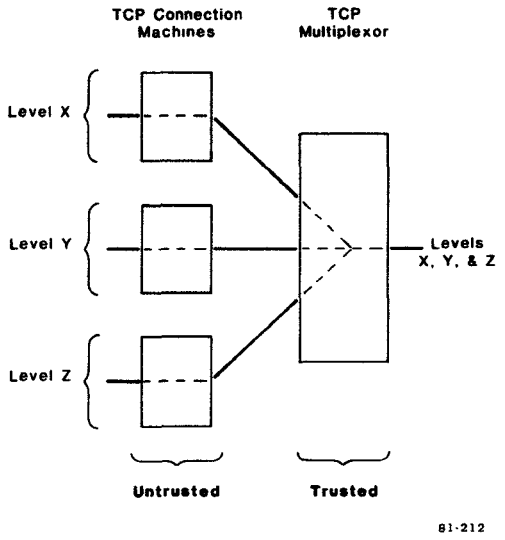
This principle is used throughout the COS/NFE to reduce the amount of software that needs to be verified. Figure 5 shows the mapping of protocols to processes for connections between the H6000 and AUTODIN II through the COS/NFE. (The figure shows multi-level connections with the H6000; this will not be the case in actual practice.) There are more processes for each connection than there were for the INFE, because both the TCP implementation and the Host to Front End Channel Protocol implementation have been divided into trusted and untrusted portions as described above.

Because there are more processes per connection in the COS/NFE, there is necessarily more interprocess communication and more process switching to perform the same communications functions. This is a property of any verifiably secure system at the present state of the art. Any operating system that is to support verifiably secure communications must therefore per-



INFE Protocol Interpreter Processes

Figure 3



COS/NFE TCP Implementation

Figure 4

form interprocess communication and process switching correspondingly faster than UNIX if the performance of the system is to equal that of UNIX.

1.3. Operating system requirements

An operating system that is to support secure communication applications should provide certain features:

(1) Low-overhead interprocess communication is required to permit the multiple layers of protocol to be implemented in separate processes as described above.

(2) Low-overhead process switching is also required to permit the protocols to be implemented in separate processes.

(3) A timing facility is required to support tasks that are dependent on real time. For example, transport and link protocol implementations employ timeout and retransmission to achieve data integrity.

(4) Prioritization of processes is required to ensure that work that is more real-time dependent is accomplished before work that is less real-time dependent. For example, processes that handle input-output hardware are more real-time dependent than processes that implement high-level protocols such as virtual terminal protocols.

(5) Verifiable security is required to ensure that data is delivered only to those for whom it is intended and that data cannot be altered by those unauthorized to do so.

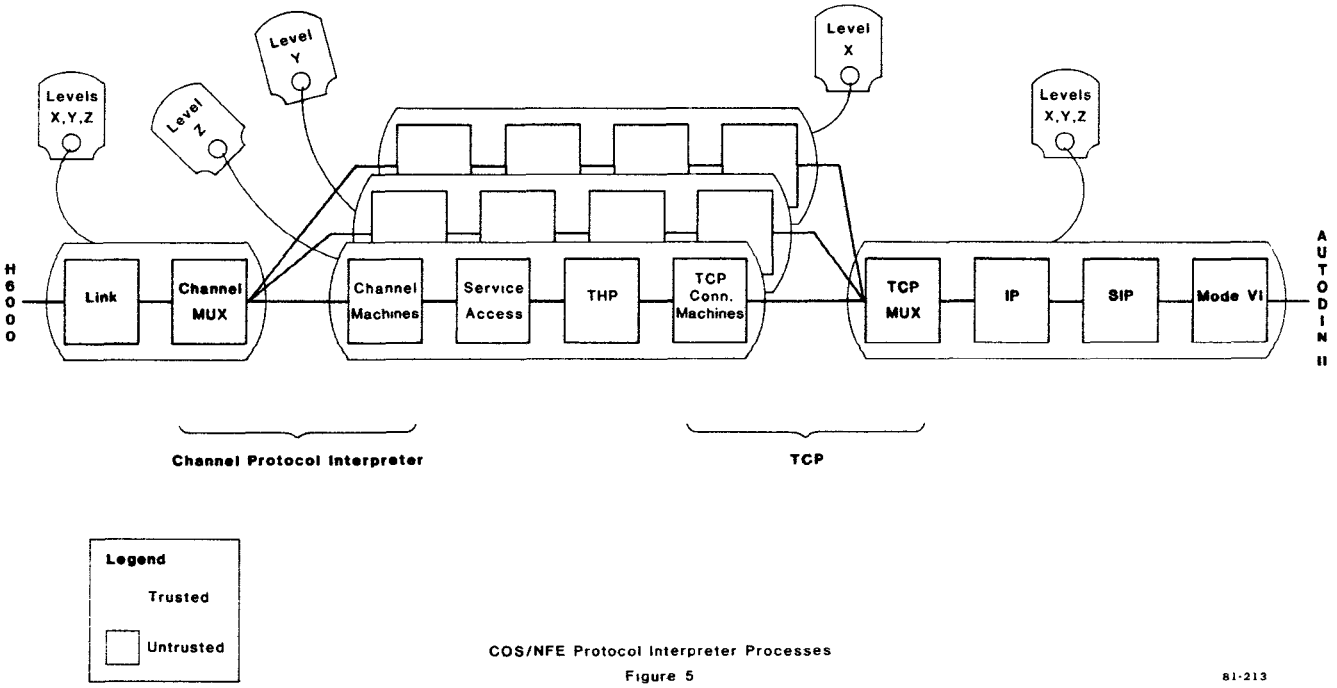
(6) A flexible security policy is required to permit the system to be utilized in a wide variety of different applications for customers with a wide variety of security needs.

In addition, the system should be portable between hardware bases to permit the exploitation of new hardware as it becomes available.

These goals cannot be achieved without an operating system designed for high performance, for security, and for portability.

1.4. State of the art

General purpose operating systems such as UNIX provide neither the performance required for high-bandwidth communications,



COS/NFE Protocol Interpreter Processes

Figure 5

81-213

nor do they provide for security. The interprocess communication and process-switching overhead in general purpose operating systems is too high for high-bandwidth communications.

There are a number of operating systems that are designed to provide security. The UCLA[8] and MITRE[9] security kernels were pioneering efforts that showed the feasibility of the security kernel approach. Their performance was considerably slower than that of UNIX. KSOS-11[10] and KSOS-6[11] are more current efforts. Their performance will never equal that of UNIX; at worst, it will be several times slower than that of UNIX. The SDC Communication Kernel[12] is designed to handle data communications functions securely, but it has not been subject to any trusted software development methodology, and its interprocess communication is slower than that of UNIX. The AUTODIN II security kernel is designed to support secure communications functions. It is written in assembler, which precludes verifying the source code at the present state of the art in software verification. It also precludes portability to processors other than the PDP-11 on which the system is implemented.

All of the secure operating systems mentioned above implement Department of Defense multi-level security as formulated by Bell and LaPadula[13]. This makes them less flexible for supporting security in non-military environments.

2. The Secure HUB Executive

The HUB fulfills the requirements for a verifiably secure operating system that supports communications applications.

(1) It provides inter-process communication functions at approximately twice the speed of INFE UNIX, which has been modified to improve IPC performance over that of standard UNIX.

(2) It provides process switching that is approximately 3 times faster than INFE UNIX.

(3) It provides timing and prioritization.

(4) Its design has been verified down to a low level of detail using System Development Corporation's (SDC) Formal Development Methodology (FDM).

(5) An analysis, also by SDC, shows that the bandwidths of any security leakage paths (potential storage and timing channels) are extremely low.

(6) It is designed for portability; machine-dependent portions of the Secure HUB Executive are isolated from the machine-independent portions, which form the bulk of the executive.

(7) It is implemented for portability; the machine-independent portions are written in PASCAL. The HUB was first implemented on the PDP-11/70. It was ported to the VAX-11/780 with a few man-weeks of effort.

2.1. HUB Functions

The HUB provides only those functions that are required to support communications and real-time applications. Functions normally found in general purpose operating systems, such as file systems, sophisticated schedulers, and swapping, are not supported by the HUB. All programs and data are memory resident.

The HUB supports processes that perform the application functions for the system. The HUB provides access to its functions through a set of primitive operations that are available as procedure calls to the processes. These functions are provided to the processes within the security constraints that are imposed by the HUB security policy.

(1) Resource management primitives provide processes with the ability to dynamically allocate buffers.

(2) Process management primitives provide processes with the ability to create and destroy other processes.

(3) Address space management primitives provide processes with the ability to alter their address spaces by mapping buffers in and out of the accessible memory space.

(4) Interprocess communication primitives provide processes with the ability to connect and disconnect from other processes and to send messages to them and receive messages from them.

(5) Flow control primitives provide processes with the ability to control the flow of messages between them.

(6) Input/output primitives provide processes with the ability to communicate to the Device Interface Handlers (DIHs) that provide access to hardware i/o devices.

(7) Timing primitives provide processes with the ability to receive notification when a specified time interval has elapsed.

These primitives cover the functions that are required for supporting communications applications.

2.2. Security policy

The security policy of the Secure HUB Executive can be described in three simple statements:

(1) Each security level is distinct from every other security level; no level includes another level; there is no order relation between levels other than equality.

(2) Security levels may be aggregated into

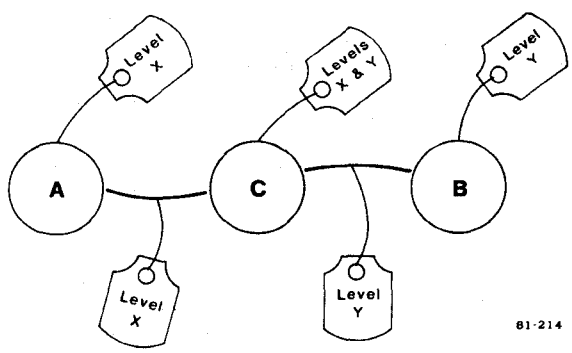
security level sets. Every process and every interprocess communication connection between processes is tagged with a security level set.

(3) Two processes that are tagged with different security level sets can communicate if and only if the security level set at which they communicate is a subset of the security level sets of both processes. In other words, an interprocess communication connection can be established between two processes if and only if the security level set that tags the connection is a subset of both of the security level sets that tag the two processes.

A process that is tagged with a security level set that has more than one member is considered to be trusted, since it is empowered to handle interprocess connections at multiple security levels. Even trusted processes are each restricted to a particular set of security levels. This is unlike most kernelized systems, in which each trusted process is empowered to handle all security levels. This restriction permits different trusted processes to handle different sets of security levels. Multiple security policies involving completely distinct sets of security levels could be implemented in the same system without interfering with one another. Using this feature, a communications processor in a public data network could handle multiple corporate and/or government customers without mutual interference.

The simplicity of the HUB security policy permits the HUB to support a wide variety of application security policies, through the use of trusted processes that make the judgements when data must be passed from a connection that is tagged with one security level set to a connection that is tagged with another security level set.

Consider, for example, the case in which two processes, A and B, A tagged with security level X and B tagged with security level Y, need to communicate. The HUB security policy does not permit A and B to be connected. But they could be connected through the mediation of a process, C, that is tagged with security levels X and Y, as shown in Figure 6.



Processes Connected via a Trusted Process

Figure 6

Suppose that the security policy that the application is to implement permits information to flow from level X to level Y, but not from level Y to level X. C, the mediating process, can implement this policy by sending to B all data it receives from A, but discarding all data it receives from B. (It could also discard all data it gets from A, but let us assume that the system is intended to do useful work.) This simple example illustrates how the HUB security policy can be used to support more sophisticated security policies, such as the Department of Defense security policy as formulated by Bell and LaPadula.

2.3. Structural concepts

2.3.1. Processes

Each process supported by the HUB consists of a program, memory, and IPC ports.

2.3.1.1. Programs

The programs for processes that the HUB supports are sharable and re-entrant. This permits many processes to simultaneously use the same program without interference. Each program is tagged with a security level set that controls the security levels of the data that it can handle.

2.3.1.2. Memory

Each process has two kinds of memory: a private memory that is used to maintain the state of computation of the process, and a set of buffers that can be used to send messages to other processes via the IPC mechanism. Private memories are not shared between processes. Buffers are serially sharable, i.e., each buffer can be accessed by at most one process at any one time. No memory is simultaneously sharable between any two processes.

2.3.1.3. Ports

Each process has a set of ports that are used to access its IPC connections to other processes. Each port is the endpoint of at most one connection. Ports are used by the HUB to keep track of the state of connections and of their security level tags. Ports are also used in buffer allocation and in flow control.

2.3.2. Interprocess communication

All interprocess communication takes place over connections between processes' ports. Connections can be dynamically made and broken. Each connection is tagged with a security level set that cannot be changed while the connection exists.

Data is sent from one process to another by placing the data in a buffer and sending the buffer over a connection. Once the buffer has been sent, it is no longer accessible by the process that sent it. When the other process receives the buffer, the buffer becomes accessible to that process.

The interface between processes and the HUB IPC facility is unique to the HUB. This interface imposes a structure on programs that is natural for implementing communication protocols, but it may be awkward for other kinds of programs.

2.3.3. Sessions

A session is a set of processes and resources, all tagged with the same security level set, that is dedicated to a particular user. For each session, the HUB controls:

(1) what programs can be executed in the session,

(2) what connections can be made by processes running in the session, and

(3) what resources can be used by processes running in the session.

2.3.4. Device interface handlers

Device interface handlers (DIHs) provide the means for processes to access input/output hardware. Each DIH performs the input/output inititation functions and the interrupt handler functions for a single hardware device. The DIHs are accessed by processes via a procedure call interface, just as in UNIX. Unlike UNIX, the DIH's execute in an address space that is separate from that of the HUB and those of the processes. This makes it possible to verify the DIHs separately from the HUB. Each DIH is tagged with a security level set that controls the security levels of the data that it can handle.

2.4. Implementing security in the HUB

The HUB is intended to enforce security in three different ways:

(1) formal security correctness,

(2) storage and timing channel bandwidth limitation, and

(3) prevention of denial of service.

2.4.1. Formal security correctness

Formal mathematical techniques have been used to verify that the HUP correctly implements its security policy. System Development Corporation's (SDC's) Formal Development Methodology[14] has been used to produce proven first and second level specifications of the correctness of the design of the HUB as it relates to security. These specifications are formal mathematical descriptions of the design of the HUB. The first level specification is very abstract and general. The level of detail of the second level specification is close to that of the HUB code itself. This can be seen from the fact that the second level specification contains approximately 2400 lines of INA JO****, SDC's formal specification language. The HUB consists of approximately 2800 lines of PASCAL. To give some idea of the magnitude of the task of proving a specification of this size, the proof runs to some 2000 pages. Fortunately, the proof process is semi-automated through the use of SDC's Interactive Theorem Prover.

The simplicity of the HUB security policy and of the HUB design makes a security correctness proof at this level of detail tractable. From a formal point of view, the most important security correctness criterion for the HUB can be stated as

(1) Let L be the set of all security levels

(2) Let S be the set of all sessions

(3) SLS: S -> 2**L

(4) All s1,s2 in S:
    s1 comm s2 <=>
    SLS(s1) ^ SLS(s2) ≠ NULL

Line 3 says that there is a function SLS that maps the set of sessions onto the set of all security level sets. Line 4 says that two sessions can communicate if and only if their security level sets have a non-empty intersection.

2.4.2. Storage and timing channel bandwidth limitation

Whenever two processes share any resource that can be modified in some way by one process, while the modification can be detected by the other process, the resource can be used to pass information between the two processes. Since the operating system is shared by all processes, even an operating system designed for security can potentially provide an information channel between processes, in violation of the very security policy that the operating system is designed to enforce. Information channels of this kind fall into two classes: storage channels and timing channels.

Storage channels are information channels that are implemented via variables within the operating system or via variables that can be accessed through the operating system. For example, if the "length" attribute of a file can be modified by one process, and if the value of the attribute can be read by another process, a potential storage channel exists.

Timing channels are information channels that are implemented via the ability to affect the time at which an event happens and the ability to detect the effect. For example, if one process can affect the time at which another process is run (perhaps by controlling the duration of its

---

**** INA JO is a trademark of System Development Corporation.

own running time), and if the second process can detect the difference between its expected run time and the time at which it was actually run, a potential timing channel exists.

It is impossible to eliminate all storage and timing channels in a system in which resources are shared by processes at different security levels. The goal of the designer of such a system is to minimize both the number of potential storage and timing channels, and the bandwidth of each.

While it is the task of formal specification and proof to show that the actions of the HUB regarding the objects that is explicitly handles do not violate the HUB security policy, other techniques are required to show that the internal mechanisms of the HUB itself do not provide storage or timing channels that can be used to violate the security policy. SDC's Shared Resource Matrix Methodology[15] (SRMM) is an analysis tool that has been used to identify potential storage and timing channels in the HUB.

The basic idea behind SRMM is to construct a matrix whose rows correspond to the attributes of resources and whose columns correspond to the HUB primitives. If the use of a primitive can modify a resource attribute, a "W" (for "write") is placed in the corresponding cell of the matrix. If the use of a primitive can detect a modification in a resource attribute, an "R" (for "read") is placed in the corresponding cell of the matrix. If there are both an "R" and a "W" in the same row of the matrix, then the resource attribute corresponding to that row can be exploited as an information channel by processes using the primitives that correspond to the columns containing the "R" and the "W". An example of a shared-resource matrix is shown in Figure 7. It is presented only to illustrate the concept; the resources and primitives shown are not drawn from the HUB.

Resource Attribute \ Primitive		READ	WRITE	SEEK	CREATE FILE	DELETE FILE
Files	Existence				RW	RW
	File Length	R	W	R	W	
	Current Location	RW	W	RW	W	
Disk Device	Arm Position	RW	RW			
	Free Space		RW		R	W

027/ORG/08-81

Shared Resource Matrix

Figure 7

The detection of a potential information channel by SRMM does not mean that there exists a storage or timing channel that can be used to violate the HUB security policy. Detailed analysis of the design and even of the source code is required to determine whether a usable storage or timing channel does in fact exist. For example, it may be that a resource attribute can be modified and that the modification can be detected only by the same process that modified it, or only by two processes that are tagged with the same security level set.

SDC performed a storage and timing channel analysis of the HUB. They first employed SRMM to identify candidate storage and timing channels. They then constructed scenarios to determine how the candidate channels could be used by processes tagged with different security levels to pass information. Finally, they estimated the bandwidth of the channels from the scenarios. The results of this analysis are summarized below.

> In every computer system with resource sharing across security domains, there are necessarily covert [i.e., storage and timing] channels that cannot be completely blocked. It is interesting to note that, in the present case, the COS/NFE design as of 9 November 1979 has relatively few such channels, as compared to other systems with which we are familiar....[16]

There are several reasons for this:

(1) The HUB is tailored to performing communications functions. The resulting relative simplicity has made it easier to control the internal resources of the HUB to limit the bandwidth of storage and timing channels.

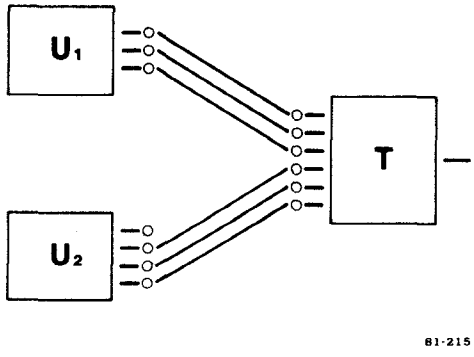
(2) The HUB has been designed with few resources shared between processes, in order to reduce the possibilities for storage and timing channels.

(3) When resources are shared among processes, the sharing is strictly controlled to reduce the bandwidth of any resulting storage and timing channels.

(4) Only specifically designated trusted processes can change resource allocation. This reduces the number of processes that can act as storage and timing information "senders".

An example of an actual storage channel that was found in an early version of the HUB is illustrated in Figure 8. Process T is a trusted process that performs some service for untrusted processes (U1, U2) at different levels. T has a finite number of ports that are intended to permit the untrusted processes to communicate with it. These ports are a resource that is shared among the untrusted processes.

Suppose two processes tagged with two different security levels are to communicate using this shared resource. One of the untrusted processes, say U1, is to be the sender, while another, say U2, is to be the receiver. To transfer data, U1 will



Trusted Process' Ports as Storage Channel
Figure 8

modulate the availability of T's ports for connection, and U2 will detect whether or not it can connect to T. U1 will connect to all of T's ports to signal a Ø, and disconnect from one of them to signal a 1. To read the value of the signal, U2 will have to repeatedly attempt to connect to one of T's ports. If a port is available, U2 will interpret the signal as a binary 1; if no port is available, U2 will interpret the signal as a binary Ø.

SDC estimated that, under worst case conditions, information could be signalled between two processes at approximately 5000 baud using this storage channel. This assumes that there are no other active processes using the system. Under more typical conditions, with other processes using the system and perhaps even interfering by themselves connecting and disconnecting to T, the bandwidth was estimated at approximately 20 baud.

Even this relatively low bandwidth requires several assumptions, some of which may not be valid:

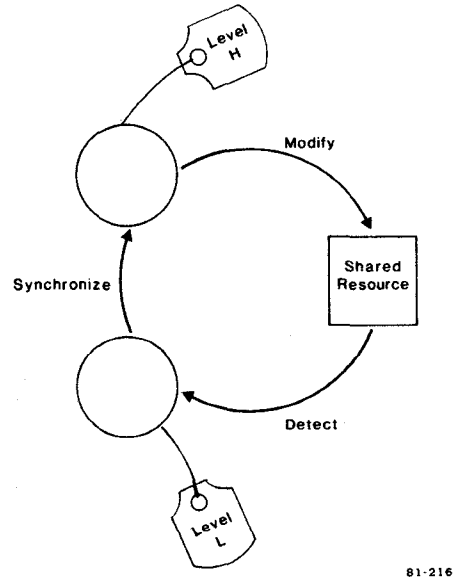
(1) The processes U1 and U2 must be able to synchronize and remain in synchrony in order to avoid confusion. This presupposes that U1 and U2 have access to a shared clock, or to some other mechanism for synchronization. The HUB does provide a facility for notifying a process after a specified time interval has elapsed, but a process is limited to the knowledge that it will not receive a "timer expired" notification before the specified time interval has expired. The receipt of a "timer expired" notification does not give the process any information about how much time elapsed between the end of the specified time interval and the receipt of the notification. Looked at another way, the clock that is available to processes is "fuzzy". This makes it difficult to use real time as a synchronization method. It also makes it difficult to exploit timing channels.

(2) The frequency of other processes' use of the ports of T must be relatively low, otherwise there will be sufficient noise introduced in the signals to make the channel useless.

(3) U1, the sender, needs to have enough ports to be able to tie up all of T's ports. This will usually not be the case, since most untrusted processes are given only a small number of ports.

Notwithstanding the invalidity of some of the assumptions, a quota was added to each session that limits the number of connection initiations that can be made from within a given session. In most applications, this quota will typically be set to a very small number such as 1 or 2 for each untrusted session. This will prevent the use of this storage channel.

The security policy of the HUB precludes one synchronization method that could be used in some systems to support a storage or timing channel. Suppose that there is some resource that can be affected by a process A with security level H, and that the modifications can be detected by process B, with security level L (see Figure 9). Suppose further that the security policy of the operating system (not the HUB) were to permit data to flow from level L to level H, but not the reverse. In this case, the operating system might allow an IPC connection between A and B, but would then allow data to be sent from B to A, but not from A to B. (This is called the "star property" in the Bell and LaPadula security model.) Then this IPC connection could be used to send synchronizing signals from B to A. This would enable A and B to use the shared resource as an information channel. The HUB security policy does not permit an IPC connection between processes under these conditions. Even if a connection is made indirectly through a third, trusted, process, the added latency will at least lower the bandwidth of any storage or timing channel implemented in this way.



Synchronization Channel Inherent in "*-property"
Figure 9

2.4.3. Prevention of denial of service

Users can deny service to other users in two ways:

1. By tying up some critical resource
2. By causing the system to fail

Users are prevented from tying up critical resources in the HUB by the same strict resource control mechanisms that are used to limit storage and timing channel bandwidth.

The likelihood of users causing software system failure in the HUB is reduced in two ways:

(1) The HUB is being subjected to a rigorous quality assurance program to reduce the probability of inherent error. This program includes rigorous testing of the correct operation of each procedure in the HUB.

(2) The HUB is a relatively simple program, utilizing simple algorithms. There is a relatively small amount of complication to "break".

3. Performance

The performance of the HUB was demonstrated in two experiments: one to show that its unique IPC interface could support complex protocol processing, and the other to show its performance relative to the INFE.

3.1. Implementation experiment

The purpose of this experiment was to ascertain whether the HUB could be used to support the complex protocol processing that is required in modern communications processors.

The Transmission Control Protocol (TCP)[17], the most complex protocol processing module from the INFE, was modified to run on an experimental version of the HUB. The feasibility of splitting up a complex program such as the TCP into trusted and untrusted portions was recognized as one key to the applicability of the HUB to multi-level secure protocol processing. The split was accomplished in a few person-weeks by a programmer who was familiar with the TCP.

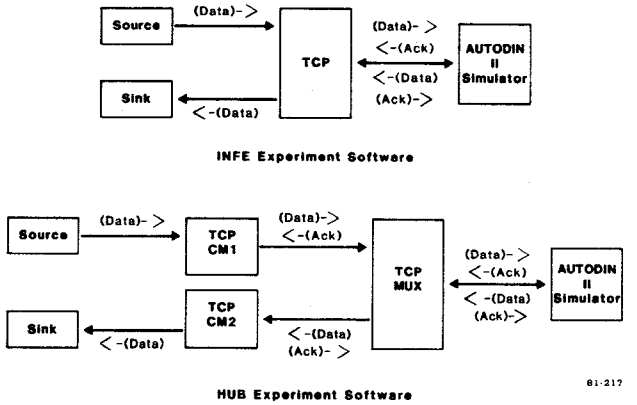
An experimental version of the HUB was written in the programming language "C" for running the experiment. Every effort was made to include the security overhead that would be present in the prototype HUB. To facilitate measurement, the experiment was run in a UNIX process, rather than on a bare PDP-11/70. It was believed at that time that the experiment would realistically predict the performance of the prototype HUB.

The TCP, both on INFE UNIX and on the experimental HUB, was driven by two processes, a data source and a data sink, respectively. A third process simulated the AUTODIN II network (see Figure 10). 2000 messages were sent from the source process to the sink process through the TCP in both implementations, and the elapsed time for each implementation was measured. The results of the experiment were very encouraging:

(1) The TCP ran 3.2 times faster on the experimental HUB than on INFE UNIX.

(2) Most of this increase was due to the great reduction of IPC overhead achieved through use of the HUB, but 15% fewer CPU cycles were executed by the TCP process running on the HUB than by the TCP process running on INFE UNIX. This is attributable to the fact that a number of functions, such as buffer allocation and timing, that had to be performed in user processes on INFE UNIX, are performed by the HUB itself. In addition, there was 40% more IPC work performed by the HUB than by INFE UNIX. Thus the experimental HUB performed more work more quickly.

(3) The source code for the TCP was reduced in size by 20%. This is also because the HUB performs functions that had to be performed by the TCP process itself in INFE UNIX.



Experiment Software Configurations
Figure 10

Based on the results of this experiment, it was expected that the overall performance of a secure network front end based on the HUB would be approximately 3 times that of the nonsecure INFE.

3.2. Performance experiment

Once the prototype HUB was coded and debugged to the point of usefulness, an experiment was conducted to see how well the prototype HUB bore out the performance expectations generated by the experimental HUB. The application level software for the performance experiment was identical to that for the implementation experiment, except for modifications that were made to accommodate differences in the user interfaces between the experimental and prototype HUBs. Again, 2000 messages were sent

from the source process to the sink process in the HUB implementation, and the elapsed time was measured.

3.2.1. Overall performance

The TCP running on the prototype HUB ran 1.17 times faster than the TCP running on INFE UNIX. An explanation of the reasons for the difference in performance between the two experiments follows.

The prototype HUB differed from the experimental HUB in a number of ways:

(1) It is coded in PASCAL; the experimental HUB was coded in C. PASCAL source code produces object code that is 30% larger (and slower) than equivalent C code. PASCAL presents the programmer with a paucity of control structures that forces tradeoffs between efficiency and understandability. PASCAL also lacks facilities (such as C pointers or the Euclid bind) that permit the programmer to avoid the reevaluation of complex addressing operations.

(2) It runs on a bare PDP-11/70 and uses the memory management hardware to effect application process protection and isolation; the experimental HUB ran in a UNIX process and simulated application process protection and isolation.

The PDP-11's memory management hardware and its relatively small per-process address space place a large processing overhead on any operating system. 24 registers must be loaded to change from one process's address space to that of another. Both the HUB and the processes it supports must execute extra code to map all the memory they need into and out of the address space. More complex data structures are required to keep track of what is mapped in and what isn't.

(3) It has a sophisticated resource allocation system that is designed to limit storage channel bandwidth; the experimental HUB had a simple resource allocation system that did not have any protection against storage channels. The prototype HUB's resource allocation scheme consumes 46% of all CPU cycles.

(4) It clears buffers when they pass from one security domain to another, and clears several machine registers whenever it switches from one process to another; these features were omitted from the experimental HUB.

The measurements were made on a first version of the prototype HUB. Very little tuning has been performed on the code. There seems to be considerable potential for improving system performance by tuning individual procedures. A few hours work on one key IPC procedure resulted in a 30% improvement in its performance.

Similar improvements can be expected in other key spots. One prime candidate for improvement is in the ring crossing code that is executed on the user side. This code currently restructures all of the parameters on each call across the system/user interface to adapt programs written in C to the HUB interface, which is designed for programs written in PASCAL. This parameter restructuring consumes about 8% of all CPU cycles.

By careful tuning of existing HUB code, we estimate that performance can be improved to between 1.5 and 2 times the performance of the INFE.

The extremely large consumption of CPU cycles (46%) by the resource allocation mechanisms is another opportunity for improving performance. In some environments, such as military intelligence, protection against the possibility of compromise via storage channels may be worth the loss in performance. But in commercial and in non-military government environments, threats involving storage channels may be considered less serious. In these environments, a resource allocator that is less stringent in protecting against the exploitation of storage channels could replace the current HUB resource allocator. A less stringent resource allocator would be simpler and would consume significantly fewer CPU cycles. The internal implementation of the current resource allocator is isolated from the rest of the HUB. Its interface has been designed to permit the entire resource allocator to be replaced.

The performance experiment did not exercise some features of the HUB that could influence performance. For example, no device interface handlers were employed. It remains to be seen whether the HUB mediation of interaction between processes and device interface handlers will affect performance.

3.2.2. IPC and process switching performance

During the two experiments, detailed measurements of the consumption of CPU time in the HUB implementation were performed. Based on these measurements, it was possible to derive the amount of time consumed by IPC operations and by process switching. In both the HUB and the INFE, the transfer of data from one process to another requires several operations such as buffer allocation, actually sending the buffer, receiving the buffer, and mapping the buffer into the process address space. The IPC facilities of the HUB and of INFE UNIX are very different. In order to compare them, it is necessary to aggregate all of these operations together and simply consider the mean amount of CPU time it takes to transfer data from one process to another in each system.

INFE UNIX consumed a mean of 41.8 milliseconds to send a message from the source process to the sink process. Sending one

message involved six interprocess data transfers, four for the message itself, and two for the TCP acknowledgement. Each interprocess data transfer in INFE UNIX took a mean of 6.97 milliseconds (41.8 / 6).

The HUB consumed a mean of 31.4 milliseconds to send a message from the source process to the sink process. Sending one message took 10 interprocess data transfers, six for the message itself, and four for the TCP acknowledgement. Each interprocess data transfer in the HUB took a mean of 3.14 milliseconds (3.14 / 10).

The HUB performed interprocess data transfers 2.2 times (6.97 / 3.14) faster than did INFE UNIX.

The time to switch from one process to another in the HUB, as derived from the experiment measurements, is approximately 300 microseconds. Measurements of INFE UNIX[18] give the process switching time as 1.175 milliseconds. The HUB performs process switching approximately four times faster than does INFE UNIX.

4. Summary

4.1. Security The design of the HUB has been formally verified to a level of detail close to that of the source code. An analysis shows that there are relatively few potential storage and timing channels through the HUB. And the HUB security policy will permit a wide variety of security policies to be implemented at the application level.

4.2. Performance HUB performance is currently 17% faster than that of INFE UNIX as specially modified for the same application. IPC operations are 2.2 times faster, and process switching is four times faster, than INFE UNIX. There is still potential for substantially increasing performance. And, by relaxing restrictions on storage and timing channels when the security environment warrants it, much higher performance could be achieved.

4.3. Portability The HUB was moved from the PDP-11/70 to the VAX-11/780 with only a few man-weeks of effort. We expect that moving it to other machines will be equally rapid.

5. Acknowledgements

The initial development of the security concepts of the HUB was accomplished in conjunction with Peter Alsberg. Steve Bunch and David Healy shared in the design and specification of the HUB equally with the author. Daniel Putnam contributed to the design of some aspects of the interprocess communication implementation and the assurance of its correctness. The coding of the HUB was performed by Steve Bunch, David Healy, Daniel Kopetzky, and Glenn Kowack. Testing and measurement were performed by the coding group with the addition of Daniel Putnam. Tom Hinke, John Scheid, and Richard Kemmerer, all of SDC, are responsible for the storage and timing channel analysis. In addition, Sue Landauer and Judy Stein, both of SDC, participated in the verification effort in conjunction with Daniel Putnam. Stephen Levin assumed the administration of the COS/NFE project, thus permitting the author to participate in the technical work. Last but by no means least, Stephen Levin, and Anne-Marie G. Discepolo of the MITRE Corporation, provided the motivation without which this paper would have, in all probability, never been written.

References

[1] Grossman, G.R., S.R. Bunch, and D.E. Putnam, COS/NFE Functional Specification, 80003.C-CNFE.5, Digital Technology Incorporated, Champaign, IL, January 1981 (Proprietary document).

[2] Bergman, S., A System Description of AUTODIN II, MTR-5306, The MITRE Corp., Bedford, MA, May 1978.

[3] Grossman, G.R., S.F. Holmgren, and R.H. Howe, "INFE Software Functional Description Overview", Document 2, Digital Technology Incorporated, Champaign, IL, March, 1978.

[4] Ritchie, D.M. and K. Thompson, "The UNIX Time-Sharing System", Communications of the ACM, Vol. 17, No. 7, July 1974, pp. 365-375.

[5] Attach I/O User Manual, 78019.C-INFE.12, Digital Technology Incorporated, Champaign, IL, October 1978.

[6] Day, J.D., G.R. Grossman, and R.H. Howe, WWMCCS Host to Front End Protocols: Specifications, 78012.C-INFE.14, Digital Technology Incorporated, Champaign, IL, November, 1979.

[7] Data Processing - Open Systems Interconnection - Basic Reference Model, Second Draft Proposal ISO/DP 7498, American National Standards Institute, New York, NY, August 1981.

[8] Popek, G.J., et al., "UCLA Data Secure UNIX", Proceedings of the 1979 National Computer Conference, AFIPS Conference Proceedings, Vol. 48, AFIPS Press, Montvale, NJ, June 1979, pp. 355-364.

[9] Biba, K., J. Woodward, and G. Nibaldi, A Kernel Based Secure UNIX Design, ESD-TR-79-134, The MITRE Corp., Bedford, MA, June 1973.

[10] McCauley, E.J., and P. Drongowski, "KSOS: Design of a Secure Operating System", Proceedings of the 1979 National Computer Conference, AFIPS Conference Proceedings, Vol. 48, AFIPS Press, Montvale, NJ, June 1979, pp. 345-354.

[11] Fraim, L., "SCOMP (KSOS-6) Development Experience Update", Proceedings of the Fourth Seminar on the DoD Computer Security Initiative, National Bureau of Standards, Gaithersburg, MD, August 1981.

[12] Colber, T., "The SDC Communications Kernel," Proceedings of the Fourth Seminar on the DoD Computer Security Initiative, National Bureau of Standards, Gaithersburg, MD, August 1981.

[13] Bell, D.E. and L.J. LaPadula, Secure Computer Systems: Mathematical Foundations and Model, M74-244, The MITRE Corp., Bedford, MA, May 1973.

[14] Kemmerer, R.A., FDM - A Specification and Verification Methodology, SP-4008, System Development Corporation, Santa Monica, CA, November 1980.

[15] Kemmerer, R.A., "Shared Resource Matrix Methodology: A Practical Approach to Identifying Storage and Timing Channels", these proceedings.

[16] Aycock, T., and R. Kemmerer, COS/NFE Security Analysis, TM-6981, System Development Corporation, Santa Monica, CA, January, 1980.

[17] Transmission Control Protocol, RFC 793, Information Sciences Institute, Marina del Rey, CA, September 1981.

[18] Jody Kravitz, personal communication.