Verifiable secure operating system software

by GERALD J. POPEK and CHARLES S. KLINE

University of California
Los Angeles, California

INTRODUCTION

While the desire for reliable security in multiuser computer systems has grown significantly, the computing community's knowledge of how to provide the requisite protection is still inadequate. Security is a "weak link" phenomenon; one link whose condition is unsatisfactory is the operating system software. It has often been pointed out that currently "no protection system implementation of any major multiuser computer system is known to have withstood serious attempts at circumvention by determined and skilled users." The community is replete with apocryphal claims of secure systems that inevitably have failed.

Out of these difficulties and concerns has grown a good deal of activity. One part of the community has addressed the questions of how a system ought to be modularized, what its primitive elements ought to be, and what logical structure would be most useful. The useful concepts of capabilities and domains have come from that activity. The Multics and Hydra systems are current examples of ongoing complex software systems in which serious attempts have been made to carefully design and structure the software with respect to protection considerations.

Despite the value of these concepts and empirical laboratories, our knowledge concerning the reliability of protection systems is disturbingly inadequate. Currently, it is not possible to provide a meaningful guarantee that a system actually provides the controlled protection which the design claims. It is not possible to state with assurance that clever users will be unable to circumvent the controls, and thereby gain access to information, operations, or other resources which the design intended to prohibit.

What is required is a system which can do more than resist attacks by penetration teams. One would greatly prefer software which in a real sense has been proven correct with respect to certain precisely stated security predicates. Such a result would provide software protection of high quality, and help relieve fears that some feature or flaw had been overlooked.

At UCLA, a multiuser computer system is being constructed in which it is expected that verification of the security properties of the software will be successfully performed. That is, meaningful and demonstrable operating system security is within our grasp. While the system is not yet complete, work has progressed far enough that the viability and quality of the resulting system and proofs seem assured. In the following, we discuss concepts which have contributed to the system's design, as well as issues that arose during that design process. It is expected that the approaches described here have more general applicability.

Note that this work only concerns the protection enforcement mechanisms of the operating system software. Properly functioning, essentially error free hardware is assumed. The problem of authentication, that is, reliable identification of the user who presents himself to the system, also is not considered, except that a suitable environment is provided in which an authentication procedure could easily operate.

In addition, the flavor of protection toward which this work is initially directed is fairly simple. Mechanisms to support mutually suspicious subsystems, memoryless procedure calls, inference, or control of statistical access are not explicitly considered. Nevertheless, the insights provided by the goals of isolation and limited sharing apply to the more complex needs.

We now discuss a number of thoughts found relevant to the design of secure operating system software.

DEFINITIONS OF SECURITY

Before continuing to discuss the general facets of the UCLA design philosophy, it will help to explain a distinction we make between security and viability. We believe operating system security involves a set of essentially negative constraints. One desires to verify that certain actions cannot occur. For example, it will not be possible for users of class x to access files of class y.

One can coherently argue that such a point of view is incomplete; the null system satisfies negative constraints vacuously. What is required, the argument might continue, is inclusion of the idea of viable productivity—that the system actually supports useful activity. For example, if a
user could cause the process scheduler not to run any processes, would this not be a security flaw? This point is well taken. Certainly productivity is of importance since it embodies the primary reason for the existence of the system.

Nevertheless, we argue that a meaningful distinction can be made between the prevention of undesired actions and the need for productive activity. To have verified the negative constraints is a useful and nontrivial step, and those constraints certainly contribute to the overall viability of the operating system.

These negative constraints express the security policy. For each user, we can translate these negative constraints into a list of which security objects this user should and should not be able to access and with what access types. Security objects are the physical and logical parts of a computer system that need to be controlled, protected, or whose status needs to be guaranteed. An incomplete list of examples includes terminals, communication lines, processes, and files.

Security policy may be expressed in terms of accessible sets. For user \( A \), \( accessible[A] \) is the set which defines what objects this user is to be allowed to access at time \( t \). All other accesses are to be prohibited. Each entry in the set is an object, access type pair.

For each active object \( D \) (device or CPU), we define \( access[D] \) as the set of accesses (object, access type pairs) that object \( D \) makes at time \( t \). We also define \( owner[D] \) as the user on whose behalf the device is performing accesses at time \( t \). For example, the CPU always acts for some given process. Then protection is enforced if \( \forall t \forall D \ access[D] \subseteq accessible[owner[D]] \). We say that a system is secure if

1. accessible sets describe the desired security policy and
2. protection is enforced.

The preceding is a simple conception of security. Nevertheless, it can yield meaningful protection and a comfortable user environment as we demonstrate below. Such questions as whether user \( A \) can steal user \( B \)'s files are, for example, directly handled by this model. Note that point one above contains aspects which will not be mathematically verifiable, since what is involved is a mapping to precise specifications from the intuitive notions which express user desires. One can check internal consistency of the resulting specifications, however.

THE CONCEPT OF AN OPERATING SYSTEM SECURITY KERNEL

The value of segregating operating systems code into more or less disjoint layers has been clear for some time. One interesting question that presents itself concerns what the lowest level of the operating system ought to contain. Designs focused around message handling, process primitives, and others have been suggested. Recent efforts concerning the “nucleus” of the operating system are illustrated by Brinch Hansen and Wulf. There is still considerable debate over this question, and its importance is not likely to decrease as systems grow more complex and additional layers, such as more sophisticated data management code, are added.

It is our contention that the very lowest system level, the subnucleus, ought to contain the security relevant portions of operating systems primitives, and nothing else. The subnucleus that contains this isolated code we call the security kernel of the operating system. Note that such a design differs widely from current practices, as illustrated by OS/360 in which that code is strewn throughout the operating system.

It has been pointed out many times that security is a crucial, basic quality of today's complex systems. The value of carefully designed modularity in the structure of operating system code is also clearly recognized. These two thoughts conjoin; enforcement of the intended modularity by security controls is of great aid in developing and understanding the system software. It is easier to obtain reliable enforcement of modularity if the proper functioning of those mechanisms which provide that enforcement does not depend at all on other code. That is, modularity enforcement code should depend on as few and as simple modules as possible. If security is basic to a system, then the code which provides security should also be basic, with few dependencies, hence at the lowest levels of an operating system.

It is exceedingly difficult to understand the interactions of security relevant software when the code and the implications of the design decisions which produced that code are distributed throughout the operating system. The difficulties arising from this distribution are illustrated by the large number of known security flaws which involve the interaction of a number of characteristics of an operating system design and implementation. As one simple example, in a number of systems, the code which checks a security condition and the code which takes the action guarded by that check are separated by some distance in the execution sequence. It is possible under certain conditions, an interrupt perhaps, for the user to change the parameters examined by the check before the action occurs. Such flaws usually depend on the availability of parameters to user code and the interruptibility of the check-action pair. As the subtleties of these interactions are seen, the near hopeless quality of poorly planned and distributed security relevant code becomes more and more clear.

The isolation and centralization of security relevant code provides more than merely a better basis for understanding that code, important as that understanding is. By isolating that code at the heart of a system, running on the bare hardware, its correct functioning does not depend on the proper behavior of other modules of operating system software. This fact provides several advantages. Outer layers of the software may be written and modified without need to reevaluate the security of the entire system. Maintenance is then eased. In addition, the resulting security kernel as a relatively small, isolated, independent set of programs is susceptible to a formal verification of its correctness with respect to security. The importance of this property is difficult to over-emphasize.
A note of caution is important here, however. It has been suggested that once the behavior of the security kernel has been verified, the security of the entire operating system software has been guaranteed. Such a statement is not necessarily true. If one’s concept of the kernel includes only security primitives, then it is quite conceivable that those primitives could be inappropriately applied at an outer level, allowing relatively subtle interactions of operating system features that lead to security flaws.

As an example, in the bulk input of a well-known time sharing system, I/O is handled by a separate process. A card deck, along with the desired file name and the target directory, is effectively given to the I/O process which loads the deck into a file and places an entry in the target directory. To do this, the I/O process has powerful access privileges. By this means, anyone may place an arbitrarily named file in any directory.

When a user executes a system command, his directory is searched before the public directory for that command name to obtain the code to execute. The interaction of this search order with the power of the I/O process results in security failure. To exercise this flaw, one merely prepares a program that performs an arbitrary action, and has it loaded into the directory of a highly privileged user. If the program is given a common command name (like SYSTAT or DIRECTORY, for example) then that arbitrary program will be inadvertently executed, with the capabilities of the privileged user, when he types the command. With care, no one will even know the error occurred. To indicate the subtlety of such flaws, it should be noted that if the search order used by the monitor were reversed, that is, if system directories were searched before user directories, this particular flaw would vanish.

In the UCLA work, this class of problems has been examined closely, and solutions to it are suggested below in the sections on virtual machines and verification. As a result, we are convinced that the kernel concept can yield certified security if it is carefully employed.

Another effective argument in favor of the security kernel approach results from the fact that it has been possible to separate security relevant code from the rest of the operating system software and centralize that code. While the verification of the UCLA system was not complete when this paper was written, the kernel isolation design had been finished for some time.

Exactly what capabilities belong in the kernel is perhaps best explained by detailed examples, but several general remarks can be made. First, the distinction of Wulf between mechanism and policy is useful. For example, while the policy contained in the scheduling process can and should be excluded from the kernel, the code which serves as the mechanism to load a process must be part of the kernel. Otherwise, parts of several processes could be loaded together (for example, the general accumulators from process i and the memory management registers of process j).

Second, the kernel can be allowed to make calls to outer layers of the system as long as the proper behavior of the kernel does not depend on those outer calls. For example, the scheduler may be called by the kernel to determine which process to run next. What reply the kernel receives is irrelevant to protection questions. The hierarchical structure is necessary for proof dependencies only, not flow of control or other behavior. This ability to allow certain outward calls is one means by which nonprotection issues can be largely excluded from the kernel code. Of course, as noted by Wulf, when the kernel calls the scheduler it is itself a scheduling decision; thus this and other questions have not been completely excluded.

### VIRTUAL MACHINES AND SECURITY

The concepts of virtual machine designs, in the sense of CP-67 or VM370, have grown in popularity and importance recently. One can view the virtual machine monitor (also called a control program, or hypervisor) as providing some basic functions of the traditional operating system, such as separation of processes, device scheduling and processor allocation. It does not enrich the process environment by providing user services. Instead, the environment produced, called a virtual machine, is essentially logically identical to the bare machine. For discussions of the value of virtual machines to operating systems design and program transferability as well as construction details, see References 7, 8 and 9.

Virtual machine designs have significant advantages in multiuser computer security systems, however, apart from the values mentioned above. One of these advantages accrues from a practical question—the amount of work required to produce a secure multiuser system. Earlier, the desirability of a security kernel was discussed. The presence of kernel code, however, changes the environment that any program sees when it is run. Operations such as changing relocation register values and direct execution of I/O may not be performed, and attempts to do so either will be impossible (when relevant areas are excluded from an address space, for example) or will abort. One is thus faced with either designing or modifying all programs to run properly with the kernel, or providing layers of code over the kernel in order to construct a more suitable program interface. The latter option could imply the building of an entire operating system.

Instead, one might layer over the kernel a skeletal virtual machine monitor (VMM). Such a task is simpler than that of building an operating system. The VMM contains no user services, and its code is devoted in large part simply to simulating those portions of the bare machine which have been usurped by the presence of the VMM itself. An elementary scheduler is of course necessary, and careful attention must be paid to I/O. Nevertheless, a VMM is still much simpler than an operating system, as might be illustrated by the relative sizes of CP-67 and OS/360. The UCLA PDP-11/45 contains certain hardware modifications to reduce the amount of supporting code for the VMM in the kernel. See the appendix for details.

Typically, user services on a virtual machine system are obtained by running a standard operating system in one of
the virtual machine environments. Thus, construction of a
VMM is a relatively simple and cheap way to obtain a clean
interface for programs, and the amount of effort required to
demonstrate the utility of the kernel is decreased.

Such an approach involves protecting and controlling entire
virtual machines. Hence two users running under an oper­
ating system in a single virtual machine are accorded no
protection from one another by the security kernel, although
they are protected from other virtual machines. The obvious
solution to this problem is to run each user with his own
operating system in a separate virtual machine, using execute
only sharing of common operating system code to maintain
efficiency.

A second advantage of a virtual machine approach how­
ever is more intrinsic, but also accrues in part from its
simplicity. Earlier, it was demonstrated with the example
of bulk I/O that, depending on one's conception of the
kernel, it was potentially possible for higher levels of software
to misuse kernel primitives in a fashion that could lead to
security flaw. This problem is diminished if the higher
levels of software are simpler.

Inter-segment linking, access control over spooling pro­
cesses, and many other operating system features are absent
in a virtual machine monitor. The complexity of the "pro­
tection semantics" associated with aspects of operating
systems such as those mentioned above, as well as the
necessary supporting code, can easily lead to security errors.
By contrast, a virtual machine system projects a very simple
environment from the point of view of security. Sharing
may be provided by very simple methods while still main­
taining the richness of an individual user's environment.

This relative simplicity makes it practical to demonstrate
that there do not exist interactions of the sort mentioned
earlier which cause security flaws. Essentially, there are
many fewer features to interact.

COMMUNICATION CHANNELS

Another relevant view of security deals with the control
of communication. For example, a system for which a user
with one clearance could not pass certain data to a user
with a lower clearance no matter how hard both try would
be of value to the military. This problem essentially is one
of showing that there does not exist a communication path
between two processes. Up to this point, we have been con­
sidering a fairly specific definition of security, albeit one
that can involve fairly subtle interactions of code. However,
as pointed out by Lampson,10 there are a number of con­
ceptual channels of flow of information in a shared system.
First, there is the obvious one that most people speak of in
the context of security, the explicit passing of files and other
units of information, reading another user's work space, and
the like.

But there are others, often involving the passing of re­
sources—assigning and deassigning devices, or making service
demands on the system that other users can sense. Exactly
what channels exist, what their respective potential band-
widths are, and what mechanisms and costs are needed to
seal them are all questions that need to be resolved in order
to decide how these communications channels should be
handled.

Of these channels, we first distinguish between those which
depend on timing considerations, and those which do not.
A timing dependent case is demonstrated by process A
sensing the CPU load of the system caused by process B.
In the context of a virtual machine system, one way these
channels may be blocked, at least in so far as the running
programs are concerned, is by properly simulating the passage
of virtual time to VMs, and not providing a real time measure.

Timing independent channels also exist, however. Con­
sider the following case. Let P1 and P2 be separate processes
who are not to communicate. The devices D1 and D2 are
dies controlled by a common scheduler S. Requests for
I/O may be considered as messages from P1 and P2 to S,
and completions as messages from S to P1 and P2. Notice
that in this example, both P1 and P2 are using both devices.

In keeping with the kernel design philosophy, it is desired
that S be kept out of the kernel. How then are P1 and P2
prevented from communicating via the scheduler S? Both
send S information, and both receive information from S.
Are we forced to prove assertions about S? If our system
already existed, its design were as above and we were at­
tempting to secure it in retrospect, without major redesign,
the answer is likely to be yes.

Instead, let the kernel do all the moving of information,
and treat the messages as being contained in "locked boxes."
A scheduler may read (but not change) such a locked box.
Label messages from P1 and P2 with their destination, D1
or D2. The return messages from devices are similarly
labelled from either D1 or D2. The scheduler now merely
queues requests and hence cannot change message contents.

SOURCES, SINKS, AND THE MORSE CODE

PROBLEM

The above design does not completely block interprocess
communication, for a "Morse Code" mechanism still remains.
That is, both P1 and P2 can pass S two types of tokens that
S can distinguish: D1 type and D2 type. Hence S can receive
an arbitrary binary encoded communication from each of the
two processes. Furthermore, S can send a nearly arbitrary
message back to each process by ordering the return of
completions (nearly arbitrary only because he may run out
of tokens of one or the other type or both). We have been
had by a malevolent scheduler. Skeptics should remember
that binary codes are the charwomen of contemporary
computing.

This problem however is not intrinsic to security systems
nor an inherent defect in the kernel concept, but rather
merely one of inappropriate design. Split the device scheduler
S into two schedulers, S1 and S2, one for each device. Now
each scheduler, if we continue to operate in "locked box"
mode, deals in only one variety of token, and hence there is
no binary communication channel.
The preceding is a rather attractive solution. The schedulers S1 and S2 need not be proven at all, and they may have full access to information necessary for scheduling. Little compromise with respect to performance has been made. The two schedulers may even share common code, as long as their writable storage is separated. The task of providing proven security has been eased, with the only important cost having been careful design.

The above solution is valid because S1 and S2 are purely information sinks. While they receive message contents, there is no way for them to broadcast, so long as only one request per process per scheduler may be pending at any time. They only request that the kernel return device status to the processes.

More generally, the solution is effective because the flow of information with respect to S1 and S2 is one way. In this case, the two are sinks. It would have been equally sufficient with respect to security if they had been pure sources. Of course in this case that would not provide the desired functionality. The principle here that does generalize usefully is that one can reduce the security problem significantly by first isolating as much as possible, often in ways that are not immediately apparent. This principle has been applied to the UCLA kernel a number of times, the device schedulers and virtual machine simulators being two examples.

At this point one might argue that these subtle channels are in certain respects irrelevant because they seem to require the active, coordinated cooperation of all the parties involved in the communication. If two users wish to communicate, let them; isn’t that the strength of multiuser systems anyway? However, in the disk scheduling example raised earlier, even without process P1’s cooperation, it would have been possible for P2 to learn about P1’s I/O characteristics. More important, security may be compromised by incorrect high level design. Errors in outer modules, such as the scheduler, are exploitable. Careful proof procedures should discover these cases.

VERIFICATION

It is important to realize that even if no security verification were actually performed, the design philosophy described here and the discipline imposed by the intent to verify, together, have already increased the reliability of the security aspects of the system. Nevertheless, there is more which can profitably be done; verification is an attainable goal.

Program verification tools are not yet in a well developed state. The largest program that is known to the authors to have been verified to date contains somewhat more than two thousand instructions; small compared to most operating systems. Hence, it is necessary to severely limit the size of the program about which one wishes to prove certain properties. The UCLA kernel is approximately 500 lines of high level code. We are in the process of verifying that code and expect to be able to complete this task with a reasonable amount of effort.

It is important to realize that program verification is not the whole of operating system security, although it is a very important part. Verification establishes the consistency of a program with a separate, essentially static statement of what that program is, or is not, to do. In order to apply verification, one first needs explicit definitions of security, and those definitions must be translated, currently by hand, into predicates that may be handled by mathematical means. Second, one is faced not by a large flowchart, but rather by a number of kernel primitives, which potentially may be invoked in any order. In addition to verifying certain properties of the primitives themselves, it is necessary to demonstrate that there is no order of invocation of primitives that would result in a security violation. If the primitives are thought of as composing an action space, then one needs to demonstrate that there does not exist a path through that space, the result of which would make one of the security predicates not true.

There are a number of strategies which can make this proof process easier. After carefully categorizing every independently addressable object and action to make the model complete, naming techniques can be employed to segment the proof task. If we can demonstrate that objects only have a user’s name associated with them if they are members of his accessible set, then we can show protection enforcement by showing he can only access objects with his name.

As a second proof strategy, certain required predicates may be included as run time checks. For example, in a military system, one might wish to guarantee that a user with secret clearance is not able to access a top-secret file. One way to guarantee that constraint is to embed a run-time check in the (only) I/O routine.

All of these considerations help to make the verification task possible although far from easy. They also support the undesirability of ex post facto verification.

THE UCLA-VM SYSTEM

Let us give some examples of the previous remarks from the UCLA-VM system. A sketch of the structure of the UCLA virtual machine system is shown in Figure 1, with the VMM broken into its attendant parts. The objects in the system have been specifically defined and are homogeneously treated and viewed by the kernel. That is, the kernel has no knowledge of the internal structure of any security objects with the exception of the protection data. This data is the basis for security decisions, and plays a role analogous to the contents of the Lampson-Graham-Denning protection matrix. It is packaged into security objects so that access to the data may be controlled. Thus, control is obtained over the way protection decisions are changed. The blindness of the kernel to other objects’ internal structure simplifies matters, but it implies that only actions among objects are monitored; no intra-object control is provided. As a result, activity within a virtual machine is not controlled by the kernel. Two users running in a single virtual machine are accorded no protection from one another by the kernel.
In order to be protected, they must each run in a separate virtual machine.

The modularity shown in the figure is needed for security. Each VM has an associated simulator. This code performs the task of simulating the virtual machine environment. However, these simulators are not proven correct. Thus to assure security of each user’s data, each simulator must be logically separate with no shared writable storage. For similar reasons, and to avoid Morse code problems (as discussed earlier), the CPU scheduler and all shared device schedulers must be logically separate.

However, to be of practical utility, some sharing among virtual machines is necessary. For example, one virtual machine contains the only ARPA network interface software, and it is highly desirable that the network be available to other virtual machines. Hence, a shared read/write segment facility between two virtual machines is included. The simulators for two virtual machines may share such a segment if so indicated by the protection data. Pseudo device interfaces to the shared segments are also provided so that standard operating systems may communicate with one another without other special facilities.

Thus, the design which is being verified supports only limited sharing, via shared read/write segments and pseudo devices. As a result, the system is rather simple and the semantics of security fairly straightforward. This simplicity has been a substantial aid to the system’s development. In addition, shared execute-only segments are also supported, so that multiple users running in multiple virtual machines but executing under separate incarnations of the same operating system can use the same code.

In Figure 1, portions of the system’s structure are highlighted. It is these portions upon which the security of the system depends. The kernel of course is included, but two modules outside of the kernel are too. The initiator and updater perform authentication as part of their primary functions. A user first presents himself through a terminal to the initiator, who eventually passes him to a virtual machine.

The updater’s task concerns the changing of protection data. It is only through this port that access to the data is potentially possible. This procedure is necessary since the state of the protection data must be guaranteed in order for the rest of the mechanisms discussed here to be meaningful. One needs a reliable channel to the kernel which does not pass through unverified VM-simulator and operating system code in order to inspect and change that data with confidence.

The UCLA-VM System is useful for practical work besides being a testbed for the development of techniques for constructing secure operating system software. The PDP-11/45 serves as an entry way to the ARPANET, replacing older equipment. The network interface software is provided by ANTS (ARPA Network Terminal System), which expects a bare machine on which to run. In addition, it will now be possible for the processing power of the 11/45 to be available for network measurements as well as local computing, not allowed by ANTS. The virtual machine monitor provides the ability to concurrently run a number of applications, each of which logically expects a bare machine, and also yields a good environment for instructional purposes, especially for “hands on” operating systems experience.

COSTS OF SECURITY

The costs of providing multiuser computer security are incurred in at least several ways: construction, user convenience, and performance. Let us consider each of these in the context of a UCLA-like system.

The cost of construction of the verified kernel with simple, understood semantic layers above it (the VMM), is certainly reasonable. The project will have consumed a small number of man years of high quality effort and no unusual equipment. A significant amount of this work would not be required if the task were repeated for another machine, and much of it is a basis for extensions.

In terms of convenience, while the system is not yet operational, it is clear from other systems that a virtual machine environment is a comfortable one for many users. The simple sharing mechanisms provide a necessary basis for network communication and inter-machine interaction.

With regard to performance, a definite answer will not be available until operational tests are made. Nevertheless, a number of remarks can be made. A definite upper bound on the performance cost of security can be obtained by comparing the performance of a procedure under the virtual machine monitor with its performance on the bare machine. Of course, that bound will also include the overhead of the virtualization process as well as limitations imposed by security.

It has been suggested that the cost of security in general may be expensive. This expense will result from overhead imposed by the necessity to follow procedures which, without security considerations, could have been obviated. Indeed, we have found places in which the procedures to perform a desired task are considerably more lengthy than they would normally be, I/O to a shared device being a case in point.

Nevertheless, we expect our security degradation bound will demonstrate that the fears of inefficiency are incorrect.
We will also estimate what portion of the observed overhead is due to virtualization costs. Hence, we confidently expect to demonstrate that a simple, but useful form of verified multiuser security may be obtained at a quite acceptable cost.

CONCLUSIONS

This work has been intended to demonstrate several points. First, and perhaps most important, it is practicable to have verified software security in multiuser computing systems. Second, the approaches of kernel design, virtual machine monitors, and mathematical verification of the properties of software contribute usefully to the task of providing verifiable security.

Nevertheless, a great deal remains to be done. It has been and continues to be a taxing effort to obtain this high level of software security for any given system, in part because our tools and concepts are still unrefined. These facts are encouraging, for they suggest that considerable progress can be expected.

Furthermore, the system described here provides only limited sharing, and does not, for example, address the problems of mutually suspicious subsystems or memorylessness at all. It has been argued that multiprogramming of resources, rather than information, is still the predominant activity, at least among the security conscious segment of the computing community. Although this segment would be satisfied by a virtual machine approach, there remain vital activities for which reliable control of sharing is crucial, and those activities are not expected to decrease in the future.

APPENDIX

The security kernel and virtual machine monitor are being constructed for a DEC PDP-11/45 that is being attached to the ARPANET at UCLA. The PDP-11/45 has a three state architecture which naturally lends itself to the needs of the kernel, VMM, and user environments. However, the Unibus I/O structure of this machine does not lend itself conveniently to virtualization, since nearly every instruction is potentially an I/O instruction and most be simulatable, unlike the limited set that normally accompanies machines with more conventional I/O processors.

However, much more important than this inconvenience with respect to I/O is the fact that the standard PDP-11/45 cannot be virtualized at all. In Reference 8, it is stated that hardware must have certain characteristics in order for a VMM to be constructible. Briefly, instructions that affect control of the processor, whose behavior are disturbed by the presence of the VMM, are termed sensitive instructions. All sensitive instructions must be privileged in order that they may trap to the VMM to have their effect simulated. In the standard PDP-11/45, there are nine instructions for which trapping is necessary but which are not privileged. This fact makes the machine impossible to virtualize.

In addition, one would like trapping of instructions to be a function of the mode in which attempted execution of the instruction occurred. The reason such behavior is desirable is a result of the following considerations. It will be natural to run the kernel in the most privileged mode, the virtual machine monitor in the next most privileged mode, and the virtual machines in least privileged mode. One then prefers that instructions which trap in a virtual machine be reflected by the hardware to the VMM, while it may be necessary that the same instruction executed by the VMM trap to the kernel. Such mode dependent trapping has been suggested before.13

In the case of recursive virtualization, in CP-67 for instance, this behavior is simulated by the software. The hardware traps all instructions to privileged mode, and software reflects some of them out. Here, however, there is an additional motivation. The existence of mode dependent trapping makes it unnecessary to have the reflection software in the kernel. It needn’t exist at all. As emphasized in the body of this paper, the need to exclude non-security code from the kernel is almost as important as including all the relevant code. The DEC/UCLA hardware modification package also includes other features for efficiency and/or convenience.

REFERENCES
