UCLA Secure Unix*

by GERALD J. POPEK, MARK KAMPE, CHARLES S. KLINE, ALLEN STOUGHTON, MICHAEL URBAN and EVELYN J. WALTON

University of California at Los Angeles
Los Angeles, California

INTRODUCTION

There has been considerable interest for some time in developing an operating system which could be conclusively shown secure, in the sense that the information stored on behalf of a heterogeneous user population was safely protected from unauthorized access or modification, even in the face of skilled attempts to do so. Early attempts to attain this goal consisted largely of auditing an existing system through attempts at circumventing the controls, and then revising the implementation code to block any successful paths that were found. Unfortunately, this approach failed to produce a secure system, largely because third generation operating systems contain so many errors that "penetration audits" followed by patches inevitably led to a system whose controls were still easily penetrated.

However, there was an even more fundamental limitation to the early approaches, frequently mentioned; testing proves the presence but not the absence of bugs. A more strictly constructive method was required, by which it would be possible conclusively to demonstrate the correctness of the security controls. It was hoped that this goal would result in a much superior system in other respects as well. The experience to be reported here strongly bears out that expectation.

The UCLA Data Secure Unix operating system is intended as a demonstration that verifiable data security with general purpose functionality is attainable today in medium scale computing systems. More specifically, the UCLA system has the characteristic that data security, the assurance that data can not be directly read or modified without specific permission, is enforced via a limited amount of kernel software. High levels of care are being applied to demonstrate that the security properties of that software are correctly implemented. In addition, the system is designed so that confinement can be demonstrated by audit of some additional, isolated code.

To achieve these goals, a number of design and implementation principles have been integrated into a single system. These include a tightly constrained base kernel, a second-level policy kernel, a well known and accepted operating system interface, implementation in the high-level language Pascal, and application of formal, semi-automated program verification methods to the source code of both kernels.

The system interface is essentially identical to Unix as released by Bell Laboratories,[9] and the software presently runs on DEC PDP-11/45s and PDP-11/70s. The kernel structures and verification procedures, together with the choice of language, provide a powerful means by which the system's security and integrity can be demonstrated and assessed. Support of the Unix interface illustrates the robustness and functionality of the resulting system.

However, the kernel and verification goals imposed significant constraints on the size, complexity and general architecture of the system. The result therefore is quite different from what would have been expected otherwise. System integrity improvement results from the significant reduction in common mechanism operating on behalf of all users, a characteristic that was necessary to make verification and certification of the system practical. Nevertheless, in retrospect, we are unaware of any decision forced by these goals which has not also had the effect of simplifying the system's structure and improving overall reliability and integrity. Significant performance penalties are not expected either. The primary cost in obtaining a secure operating system appears to be found in the care required during design and development.

In the next sections we outline the UCLA Unix architecture, together with explanations for the design choices. Verification and the programming language are also discussed, and illustrative examples of the effects of Unix functionality on the system's operation are given.

OVERALL ARCHITECTURE OF UCLA UNIX

The UCLA Unix architecture contains a number of major modules, whose relation to one another is suggested by Figure 1. The kernel should be thought of as an operating system nucleus which provides about a dozen primitive operations callable from user processes. That is, the kernel implements a number of abstract types and the valid operations on each type. It is the only module in the system empowered to execute hardware privileged instructions.

* This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-77-0211.

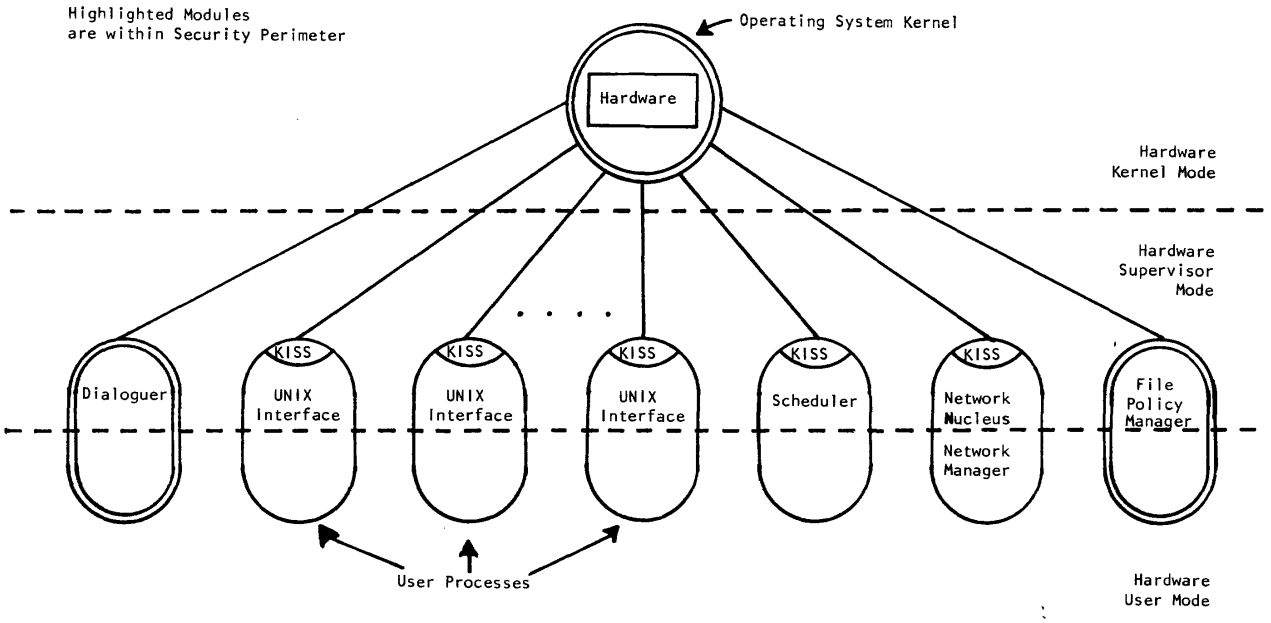

Figure 1—UCLA Secure Unix architecture.

One of the abstract types implemented by the kernel is process. A process contains two address spaces (supervisor and user mode on the large PDP-11s). An operating system interface package resides in one address space. In the other, application code is run. When an application program makes an operating system call, control passes to the O/S package which interprets the call. If necessary, the package issues kernel calls or uses kernel facilities to send messages to other processes to accomplish the needed action. All such calls or messages are controlled by the kernel. Each process is a separate protection domain. The access rights of the domain are represented by capabilities: a C-list for each process is maintained by the kernel.

There are several processes that are special, in that they perform system-related functions. Overall system security depends on the correct operation of two of them.** One, called the policy manager, is the process capable of altering the data upon which protection decisions are made, and is thus the site where various security policies may be implemented. Type extensions to kernel objects, including file systems, typically would also be supported here. In the UCLA system, security policy plus suitable primitives for the Unix file system to support protection of individual files are built in the policy manager process. The second, "dialoguer," process initially owns all terminals (i.e. has capabilities for all of them) and is responsible for user authentication. It tells the policy manager what user is to be associated with a given process.

There is one further process which differs from the typical processes employed for applications programming. However, this one, a scheduler, is not relevant to data security. It contains short-term resource management policy for CPU and main memory: process scheduling, page replacement strategies and the like. UCLA Unix is a demand paged system; when a process page faults, the scheduler is informed by the kernel so that an appropriate swap call may be issued at some later time by the scheduler. All of its security relevant actions are accomplished through kernel instructions, however.

Thus, in normal operation a user first logs into the dialoguer. That process then sends a message to the policy manager, who initializes a process for the user and moves the user terminal to the new process by issuing appropriate capabilities. Process initialization as well as normal computation take place within the domain of the given process. Additional resource requirements or file activity is accomplished through messages to the policy manager. Process switching occurs whenever a given process invokes the scheduler process, or when an appropriate clock interrupt forces such an invoke. The scheduler can then run whatever process it wishes. Page faults also force an invoke of the scheduler, so that it can initiate appropriate page swapping.

THE UCLA KERNEL AND ABSTRACT TYPES

The kernel can alternately be viewed as a basic, stripped down operating system or as an implementor of a number of abstract types, together with the operations on those types. One of its more notable features is the fact that a significant number of facilities, normally found in large systems, are included in it despite its very small size and straightforward structure. The basic kernel consists of ap-

** One might say they are within the "security perimeter." Their size is not large compared to the kernel described here. It should be emphasized that the design is such that only software within this security perimeter must be correct to block system penetration.

proximately 760 lines of Pascal code, not including I/O support. The PDP-11 does not have any channels, so that the functions of channel programs must be written as CPU code. I/O support in the UCLA kernel is composed of two portions: a device-independent internal interface of approximately 300 lines, and as many device-dependent drivers as are required by devices present on a given machine configuration. These are quite small, and for the UCLA installation, supporting many peripherals, approximately 300 lines of code are required altogether. These numbers are relevant because it is intended the entire kernel be subjected to program verification procedures. Given current verification capabilities, and well structured code, this goal is attainable.

The UCLA kernel implements a fixed number of types, the four listed below. Type extensibility as illustrated by CAL-TSS or Hydra is not provided, although simple extensions appear feasible. The implemented types, together with the permitted operations, are discussed below.

Processes

The process object is defined to consist only of the usual state variables plus one small page. It does not include the process virtual memory. As a result, kernel calls such as Invoke can be quite simple, merely moving data from tables to CPU registers and vice versa. All process relevant kernel calls are controlled by capabilities. It is not possible to send or receive interprocess communication, for example, unless in each case a capability is present in the process' C-list.

The process abstraction has been carefully developed to permit a large number of processes to be alive—500 on a PDP-11/70 would not be unreasonable. To do so, it is necessary that very little locked down memory be required per process, despite the fact that there are asynchronous events taking place (such as I/O completions and messages from other processes) which can occur when all the memory of a process is swapped out. The process must be informed of these events. However, the obvious solution, kernel queues, are undesirable since they increase verification difficulties, lead to overflow problems when queue space is exhausted and introduce confinement problems. The UCLA kernel avoids this problem by a number of methods, including a generalized page faulting structure and efforts to keep as much per process information as possible in swappable pages allocated to the given process. As a result, very little main storage must be reserved for an active process. Further, the complexities in two level process mappings, as suggested for Multics are avoided.

The operations available for objects of type process are as follows:

a. Invoke
b. Initialize
c. Zero-relocation-register
d. Return
e. Send-interrupt***
f. Set-interrupt***

*** These operations are available for all objects.

Invoke moves the state variables of a process into the CPU registers, after first saving those of the currently running process, mostly into one of that process' pages. Initialize clears the state variables of a process and grants those few capabilities needed for the process to bootstrap itself. The Zero-relocation-register call is used by the process to adjust its virtual memory. The process first sets a data structure in the page shared between it and the kernel to indicate the desired virtual memory page, and then zeroes the associated register, so that the subsequent page fault will cause the kernel eventually to reload the register with the desired value. Return is used by a process to change its current site of execution, either giving up control to the scheduler process, checking to see if any interrupts have arrived, or changing from supervisor to user mode. Set-interrupt and Send-interrupt are used in the system's inter-process communication facility. Both calls give as a parameter the name of an object with which the signal is to be labelled. Set-interrupt is used by a process to enable its ability to receive signals labelled with the named object, and send-interrupt is the associated inter-process signalling mechanism. Send-interrupt also passes a small amount of data, and is the means by which the system supports very low delay inter-process communication (ipc). High bandwidth ipc is done through shared pages together with this interrupt mechanism.

Pages

Pages are the abstract storage unit supported by the kernel. All pages have a fixed home location on secondary storage, which is not deallocated when the page is swapped into main memory. There are two page sizes in the current implementation, with memory frame sizes set at sysgen time to minimize kernel complexity. In order to access a page, a process must first obtain a capability for the page. Then the process indicates where in its virtual address space the page specified by the capability is to appear. At that point the process can attempt to refer to the page. If it is in core, the hardware register will be loaded and the reference will succeed. If not, the process will page fault, as will be described. Since each page is a separate object, controlled sharing of individual pages is easily done.

The only operations on pages are:

a. Swap-in
b. Reflect
c. Free

"Swap-in" copies the secondary storage version of a page into main memory, changing the name of the object associated with that destination page frame to the new page. The secondary storage copy is preserved. "Reflect" updates the secondary storage version to match main memory. Neither of these operations gives the caller access to the contents of the page, so that the operation can be issued by untrusted code. "Free" deletes a page from main store (only permitted if the secondary storage version is current.÷)

÷ This (trivial) operation is present only so that pages can be moved from place to place in main store. Without "Free," this movement would be more difficult, since the swap call nops when issued if the specified page is already in main store.

To simplify management of physical memory, main memory is divided into two statically allocated regions, one for each page size. That is, one contiguous region in main memory is devoted to small page frames, the other to large page frames. The portion of main memory allocated for each size can be varied at system generation time by changing a compile-time constant.

A similar situation exists for disks, except that the number of regions is variable and changeable at kernel system generation time. A given physical disk unit is broken into an arbitrary number of smaller logical mini-disks. A mini-disk is constrained to contain a fixed number of pages of a given size and object type. Mini-disks, however, are not required to be of uniform size. Hence one mini-disk might have X small pages, while another mini-disk has Y small pages. While large and small pages are not allowed to co-exist on the same logical mini-disk, they can easily exist on the same physical disk. This structure simplifies address calculation within the kernel for pages on disk, yet still allows the system manager freedom to rearrange the configuration of large and small segments on the disk to minimize disk seek and transfer times.

Pages have a home address on disk, and the name of a page can be used to determine this home address in a simple way. Main store is simply a disk cache; in-core pages are therefore merely copies of disk pages.

Devices

I/O operations to all devices, including terminals, are controlled by the same capability mechanism as all other operations. However, devices such as terminals are treated as two devices—an input part and an output part. Two capabilities are therefore required to read and write a terminal, but as a result kernel internals are simplified.

Completion interrupts are handled just like any other process notification. All those processes with capabilities to receive interrupts from the device, with interrupts enabled, and with appropriate access, will receive a notification when the device generates it.

The device operations are as follows:

a. Start-i/o
b. Completion-interrupt
c. Status

"Start-i/o" initiates all I/Os except swaps and reflects. The "Completion-interrupt" is the hardware generated call which typically signals completion of a previously started I/O. As an entry point into the kernel, it is little different from any other call. "Status" I/O causes no input/output operation, but returns status of the previously started I/O.

Capabilities

The capability is the basic kernel representation of protection information—which objects a process is entitled to access. Each process has associated with it a C-list containing those capabilities, stored in pages that can be swapped, but which are directly accessible only to the kernel.§

Each capability consists of four fields. First is the name of the object to which this capability refers. Second are the access rights provided. Next is a "guess" value which the kernel uses to attempt to quickly find the entry in a kernel table which maps the object indicated by the capability to a physical location. In the case of pages, the guess is the index into the kernel page table to the slot where that page entry last appeared. It in fact may have been moved by subsequent Swaps and Reflects, so if the entry does not match, a search of the table is required. That event is relatively rare however. The last field in the capability is of no relevance to the kernel, but can be set via the Grant call. The Policy Manager uses it to record the file descriptor with which the page or device is associated.

The only operation on capabilities is:

a. Grant/revoke

It adds a specified capability to a specified slot in a specified process' C-list. What processes can issue this call is also controlled by capabilities.

The operations on capabilities are thus quite limited. Revocation is accomplished by granting the null capability into the C-list slot that contains the capability to be revoked. There is no means by which processes can directly pass capabilities. While this fact limits what can be done with capabilities, it also greatly simplifies many issues and avoids a number of the criticisms of certain capability systems, especially the danger of not knowing how access to an object has propagated. As a result, the kernel can more accurately be viewed as containing no security policy. All such decisions regarding rights transfer, including initial granting of rights, are made only by the software running in the process which has the ability to issue Grants. The Policy Manager is the only such process in UCLA Unix.

The C-list composes a local name space for the process. This name space has two effects. First, through message exchanges with the policy manager, the user has complete control over which C-list slot contains a given capability, thereby permitting local management over the name space. Fabry[1] points out the significant advantages of this facility. Second, kernel names are not visible to user code. Instead, the capability contains that name. Therefore user code, being unaware of the actual object names, cannot use them as a means to breach confinement.

Types and operating systems

Other authors[11] have noted that the usual views of abstract types to be found in programming languages are not quite suitable for operating systems because of finite resources

§ The policy manager is given read access to capability pages so that it need not keep separate track of which capabilities for pages in a file are outstanding. See the discussion of the policy manager for further information.

and circular dependencies. In Multics, for example, the process manager depends on the page abstraction, since the manager is contained in pages, while the page manager is a process and hence depends on the process manager. In a revised design for Multics, abstract types are used in a sophisticated, multiple layered manner to solve these problems.[11]

However, as noted by Gaines,[2] the method required need not involve a sophisticated solution at all, and is largely composed of static allocations. This is the approach embodied in the UCLA kernel. Processes, pages and devices are neither created nor destroyed. There are as many pages as there is space on secondary storage for them. The number of processes is fixed by the size of the kernel process table. Devices are added at system generation time. This static view is not really a limitation, since the Policy Manager reuses process "bodies" and pages by reinitializing them via kernel calls. Many systems include these size limitations anyway, although perhaps not so explicitly. As a result, the kernel type structure is exceedingly simple, and yet robust enough for fairly general operating system activity, as illustrated in the sixth section on Unix functionality. Further, the entire kernel is small enough to be locked down in main memory, in space removed from page management, blocking circular dependencies.

Kernel names

The names for kernel-supported objects were designed to maintain several important properties with the minimum of mechanisms: a) Unique names for all objects, b) clear knowledge of object types at all times, and c) avoidance as much as possible of complex name to location mappings, which must be maintained by kernel code if object protection is to be at all meaningful. Since these names are not visible to normal user processes, who see only C-list indexes, considerable design freedom was present. Therefore, names were chosen to represent the home location of the object: a page name consists of the disk device and block number. Hence no disk map need be maintained or interrogated by the kernel.

Paging, segmentation and scheduling

UCLA Unix, unlike standard Unix, is a demand paging system. All user disk I/O, including swapping of the process virtual memory space and file activity, occurs via the paging mechanism.*

Page faulting is invisible to all processes except the scheduler, which is invoked by the kernel when a fault occurs, so that it can start a swap. There are actually two "faults" involved in accessing pages. The most significant, just described, occurs when a page is not core-resident. The other, called a register fault, occurs when the page is resident but the relevant page register is null. This case is handled in a highly efficient way—a user map table is checked by the kernel to see which capability (and therefore which page) is desired. The appropriate value is then placed in the register and user execution continues.

All kernel calls which require swappable pages to be in core in order for successful execution first check to see if the necessary pages are indeed available. If not, the call completely unwinds itself (a trivial act, since no kernel table updates are made until all checks complete successfully), the process state is reset as described above, and the scheduler is notified as in any other page fault. Even invoking a process, which requires the page that contains the user's registers, operates in this manner. Thus page faults involving kernel-primitive instructions appear to user processes just as page faults involving hardware implemented instructions (that is, they are completely transparent).

The preceding outline suggests how the UCLA system provides a complete virtual memory and file system with only a simple set of paging primitives in the kernel. This simplicity was achieved by two major decisions. First, the virtual memory facilities were decomposed into that which had to operate correctly in order to maintain the security and integrity of the system (Swap, Reflect, and Completion-interrupt) and the rest of the virtual memory mechanism (page replacement algorithm, interaction with CPU scheduling, etc.). This decision had a significant effect on the system's resulting simplicity. Second, file activity and process memory swapping were combined into one mechanism. In standard Unix, main memory is broken into two areas—one to hold user process images, and the other for I/O buffers. Each area is managed separately. The I/O buffers are replaced in LRU order, while scheduling of process images is handled differently. All disk I/O buffers are the same size, while process images vary. The code used to handle I/O buffers is in large part different from that used to handle the movement of process images, and significant parts of both collections of code are important to the system's security and integrity.

In UCLA Unix, only one mechanism, paging, exists, and much of its support has been moved out into a scheduler which can not affect the integrity of the system. As explained earlier in the section on capabilities, the user domain also carries some of the responsibility for virtual memory management. By placing some of the responsibilities in the domain for which the action is being taken, error propagation is further limited. Application code is of course unaware of that responsibility, since the O/S interface is performing the task.

Firmware implementation

The UCLA kernel has been developed to be a candidate for firmware implementation. To be practical, it is helpful if each call behaves as much as possible as a separate instruction, with no need to be interrupted in execution, nor to issue I/O calls for which the results affect the instruction's behavior, since I/O is typically slow relative to micropro-

* A physical disk can alternately be treated as a device, and Start-I/Os issued to it. However, a disk treated in this manner cannot also hold pages.

gram cycle speeds. These criteria are met by the UCLA kernel. Therefore, it differs significantly from architectures such as Multics or related work.[6,7] In both of those systems all of the operating system, including inner rings in Multics and kernel software in the case of MITRE, must be considered as part of the user process. Any process can be suspended in the middle of execution in the inner ring or kernel mode, respectively. Neither of those systems lend themselves to firmware considerations, the MITRE work because of the architecture, and Multics because of its size and architecture.

Verification impacts

Verification of a full scale operating system is a multistep process, and the methods employed at UCLA are outlined by Popek,[8] with more detail available from Kemmerer.[3] The effect that the verification and certification goals had on the system architecture was exceedingly positive. Often a design choice presented itself, without any clear basis for resolution except maximizing verification ease. In retrospect this criterion was quite effective in making decisions and avoiding design pitfalls. Further, when it became clear subsequent to implementation of certain parts of the system that verification would be difficult, those portions were redeveloped. A good example of this case will be outlined in the section on I/O Interfaces.

Sequential code

The current state of verification tools does not permit proof of parallel programs. Since semi-automated aids are, in our view, essential, this constraint implied a kernel design and implementation in which each call ran from start to completion without interruption, including the interrupt handlers. The UCLA kernel is built in this way, and so most of it can be proven by standard verification methods.

The cost of this design choice results from delayed servicing of interrupts which arrive while a kernel call is in progress. To minimize this problem, each call is designed to run very quickly—approximately one millisecond or less. To do so, no kernel call may do I/O of its own while in the midst of execution, since virtually all devices respond rather slowly relative to this criterion. While millisecond delays in interrupt servicing may not be suitable for heavy real time activity, it appears quite acceptable for interactive systems, which is the nature of Unix.

I/O Interface

The PDP-11 does not have any significant channels; instead the device registers are wired into physical address locations and "channel" functions are executed by CPU code. Since all devices address main memory (and secondary storage) in terms of absolute addresses, I/O management is therefore necessarily a kernel responsibility. This is unfortunate, for several reasons. First, device semantics are quite complex and difficult to interface with the semantics of the programming language in which kernel code is written. Next, devices are probably the single largest source of changes to the kernel, since as new types of devices are added, additional verified kernel code is required to manage the device's actions. To minimize the impact of these problems, kernel I/O code was redesigned to provide a device independent level of I/O abstraction within the kernel. Code above that level is not concerned with any of the device details. Code below it implements device dependent issues, including any device dependent protection controls. The I/O abstraction level appears similar to a channel interface, with well defined opcodes and operands.

This I/O abstraction level is quite important, likely more so than the process abstractions mentioned by other authors, since at least half of the operating system kernel is concerned with I/O.[11,6] As a result of its use, device semantics have been isolated to the low-level drivers. See Walker[12] for more information.

THE POLICY MANAGER

The Policy Manager is the major security relevant process in UCLA Unix. It is responsible for implementing a shared file system, for maintaining whatever security policy is to be supported by the system, and for part of the action of process initialization, which occurs every time a Unix fork operation takes place. Each of these issues is discussed below. Long term resource allocation can also be implemented in this process, but currently is not.

The file system and protection policy

User code must see a file structure which is identical to the Unix tree of directories. However, one should not immediately conclude that the entire directory structure and other file support should be implemented in trusted code. In fact, one can make the following argument, largely independent of the security policy to be enforced.

Most code to be run in the user domain strictly should not be trusted to be correct, at least not to the same standards as the verified secure kernel and policy manager. However, all names, including file names, are either issued, interpreted or transmitted through that code. Therefore, it makes little sense to verify the directory-naming scheme of a file system when significant amounts of unverified code issue the names or are in the path leading to the file system. The best one can do, it appears, is to provide the user with a reliable means to specify a process profile which characterizes the categories of files to which the process is to be allowed access. Profile specification and alterations, together with the association of labels with the file on which categories are based, must therefore be done in a guaranteed reliable way if the verified protection and integrity of the entire operating system is to have any meaning. That necessary secure terminal facility will be discussed in the seventh section.

The file protection labels provided in UCLA Unix consist of a very large variety of "colors." Each file can be labelled with some number of them. Each user (principal in Saltzer's terminology[10]) has a fixed color list associated with him. It is understood that a user potentially can access a file only if his color list covers that of the file. The actual profile for a running process can be set to any subset of the user's color list. There is a separate profile for read and write.

Since there are a large number of colors, many of the usual protection policies can be implemented using them. Public files are labelled with the color public and all users have that color in their list. Denning has noted that military security policy is essentially a lattice, and that the relations of sets and subsets provides just the lattice required. Individual file names are had by assigning a given color to a single file. This color system is still evolving as experience is gained with the user protection interface, especially in the area of control over changes to color lists. Additional detail is provided by Urban.[14]

Given the preceding view of file system protection, one can profitably decompose its implementation into two parts, one a common mechanism relevant to security and integrity, the other executable in the domain of the requesting user process. The common mechanism can support a simple, flat file system. Files are the only significant data type, and a color list is one of the attributes of a file. The simple file system mechanism must include complete space management—disk-free lists and maps specifying which pages belong to which files, together with software to manage these data structures.

Many of the facilities normally thought of as part of the file system can be provided by software in the individual process domains as part of the O/S interface—directory structure, maintenance and searching; end of file indicators and other file status information such as usage locks. Directories are then contained in files, and access to directories is controlled in the same way as access to any other files. Assuming that the common mechanism in the policy manager is verified correct, users can affect one another only through the use of files to which they share access. Once again one expects system integrity to be further enhanced, as errors in higher level file system code are confined to the domain in which they occur.

Process initialization and forking

The policy manager must also be involved when new processes are created, since a kernel process body must be Initialized and appropriate capabilities need to be granted to the new process. As much as possible however, one wishes process bootstrapping to take place within the domain of the new process. In UCLA Unix, the normal procedure for process forking is as follows. The requesting process sends a message to the Policy Manager requesting the new process as a member of the same user family. The Policy Manager records the user to be associated with the new process and issues a kernel Initialize call, which zeroes a process body, grants two capabilities to that process, and sets the program counter and status to standard values. The capabilities point to a standard boot code page and the arg-block page respectively.** A third capability is granted by the policy manager upon process request to give the process the ability to communicate with its forking parent. From here on, initialization takes place wholly in the domain of the new process. The process begins by attempting to execute its boot code, which may cause a page fault. These are handled normally. Eventually the boot code will load the O/S interface and presumably a Unix Shell into its address spaces.

Other policy manager responsibilities

In UCLA Unix, the Policy Manager is also responsible for control over access to the other kernel-supported objects besides pages—processes and devices. Devices appear as special files and inter-process communication takes place through pages which appear as part of a file. Therefore, colors are uniformly employed for access control in these cases too.

An ARPANET connection is provided in UCLA Unix; access to it must be controlled and support for initial network connection activities is required. Capability based encryption is used to protect each connection individually. See the section on Secure Computer Networks.

THE KERNEL INTERFACE SUBSYSTEM

Since the kernel is an operating system nucleus of minimum size and complexity, one can properly expect that it is not a convenient base to build on. Traditional systems provide a good deal of "extension" for convenience. While at first glance the O/S interface has this responsibility, it should be noted that a considerable amount of code is written to run directly on top of the kernel—the O/S interface, the network manager, process initialization, and the scheduler, for example. Each of these need basically the same extensions—capability management, inter-process communication support, virtual memory code, and some file system interfaces. Therefore we have developed an intermediate interface between the O/S interface and the kernel. The software which implements it provides a much more convenient interface to the kernel and is called the Kernel Interface SubSystem (KISS). As an extension mechanism, the KISS manages the entire environment of the process. In general, no other code in the process makes kernel calls, sends messages to the scheduler or policy manager, etc. Thus this software package has primary responsibility for maintaining a convenient "virtual machine" for the user process.

The KISS of course runs as part of the user process domain, and is architecturally contained in the same address

** The boot code is actually the Kernel Interface SubSystem discussed in the fifth section. The arg-block page is read/write shared between the process and the kernel, and serves as the means for passing arguments and return values for kernel calls.

space of the process as the O/S interface. The KISS can be viewed as an inner ring in the sense of Multics, and if appropriate hardware were available, that would be an effective means of implementation.

THE UNIX INTERFACE

The operating system interface has the responsibility of providing a user program interface which is as much as possible identical to standard Unix.*** It handles user system calls either by performing them itself if possible, or making the appropriate kernel calls for service requests to the policy manager to get the desired action accomplished. Parts of the Unix O/S interface are actually composed of code from the standard Unix operating system. Most of the changes consist of wholesale deletions of functions, resulting from the fact that many of those functions are redundant given the available kernel facilities and the fact that the O/S interface is essentially a single user system. All scheduling support could be removed, since scheduling is done in a separate process. A more drastic change concerns I/O buffering. In standard Unix, buffers contain significant structure to aid in multiuser and LRU operation. In UCLA Unix, most of that function disappears since it is done by the paging mechanism supported by the kernel and scheduler. I/O support is replaced in the O/S interface by code that requests file opens and relevant page capabilities from the Policy Manager, and maps those pages to the interface's virtual memory. Then the interface merely tries to reference data on the page to move it to the user, and the usual page faulting and swapping action takes place.

New code in the interface largely consists of the KISS, changes to the interface/KISS boundary, ipc support, and maintenance of the process hierarchy. This last issue will now be discussed.

The file system

The Unix interface has a significant portion of the responsibility for making the user view of the file system equivalent to standard Unix. This task consists of all directory support, including searching, working directory control and the like. Once the desired logical file name is found in a directory, a file open request of the policy manager can be made using that name.† Directory searches are done by first opening the containing file, like any other. It is the responsibility of the Unix interface to manage its open files in such a way as to keep the working directory open most of the time to minimize search costs.

*** There are certain actions possible in standard Unix which will be blocked by the security policy of the secure system.

† The logical file name is essentially an inode number. (Pointer to file descriptor in the Unix file system.)

Forking and process hierarchies

In standard Unix, a given user can have a process family active for him. The family is hierarchical in the sense that parents have certain rights over children. However, intra-family protection is not really effective, since any member of a family can convince any other member to destroy itself, and to take other undesirable actions, via standard Unix functions.

Therefore, process hierarchies should not be supported by kernel code, and so in UCLA Unix, members of a process family cooperate among themselves to effect family behavior. Of course, the support for process families is provided in the O/S interface, so that user software need not be concerned. This design choice simplified the kernel, and in light of the observations just made, had little or no effect on the actual protection functionality provided.

In the implementation, each process of a family has a capability for a shared page, set up by family members. In that page, data structures are maintained by the O/S interface so that intra-family relationships are properly supported. In doing so, the kernel notification facility is used to great advantage. Unix typically performs a great deal of "one to n" notification—one process issuing a signal intended for the rest of the family. The kernel Send-interrupt call is designed to support this behavior efficiently, as well as to be adaptable for other uses.

SECURE USER INTERFACE

In order for any user to have assurance that the protection controls of a system are operating in the manner desired, it is crucial that he be sure of the values to which protection policy data have been set. Further, when login takes place, there is an issue of mutual authentication: the user wishes to be sure that he is interacting with the secure system interface, not some clever user simulation of it which collects passwords. For both of these reasons, UCLA Unix contains a small dialoguer process to which the user terminal can be reliably connected. The user causes his terminal to be switched to the dialoguer by typing a predefined sequence of break characters.§ The kernel supports the terminal switch through maintenance of a terminal state. A terminal can be thawed or frozen. Capabilities are granted by the Policy Manager giving access to terminals only when thawed, or only when frozen. When the break sequence is detected, or when a line drop occurs, the line is marked frozen. The Policy Manager grants frozen access only to the dialoguer, thawed access in all other cases. In this way, the user can move his terminal to the dialoguer, accomplish whatever change is desired, such as changing process profiles, and then move the terminal back, all without disturbing the state of computation of the process at all so that it can be continued.

§ Kernel recognition of the break sequence is not expensive since PDP-11 hardware requires character by character terminal input handling anyway.

THE SCHEDULER

Whenever it is time for a process invocation decision to be made, the Scheduler is invoked, either directly by a user process (i.e. when it wishes to sleep) or by a clock interrupt. The kernel makes available a considerable amount of system data through a pseudo device, so that the scheduler can make sophisticated resource allocation decisions, about both memory and the CPU. Centralizing both classes of resource control permits effective coordination of allocation decisions and therefore potentially higher performance. A large class of scheduling policies can be implemented in this process. Some of them have confinement implications but provide better performance potential than those which do not. This architecture permits the system operator to make the confinement/performance tradeoff, since there is no kernel effect from scheduling policy changes.

The one potential drawback of a separate scheduler process is that it doubles the actual number of process invocations over what is really needed. This overhead is of little consequence if context switches are relatively cheap, not really the case for UCLA Unix.*

SECURE COMPUTER NETWORKS

When security is of concern in a computer network, encryption of the lines is generally a necessity, because those lines are not considered safe from tapping or spoofing. However, the usual approach is to encrypt and decrypt the data external to the central machine and its operating system.

It should be recognized that the software resident within the operating system responsible for managing the network is both complex and relevant to security and integrity. In standard Unix with an ARPANET Network Control Program (NCP), the NCP, operating as a common mechanism, is of comparable size and complexity to the whole operating system.** Typically, one wishes to protect each network connection separately from each other connection, but the NCP manages them all, including moving data from user buffers through the NCP and out to the network interface device.

Given the availability of a secure operating system, one can entertain the idea of extending the "ends" of the encryption path deep into the operating system. For example, the user process, as it hands data over to the NCP, could be forced to cause the data to be encrypted, so the network software is treated merely as part of the insecure transmission channel. That data would not be decrypted until the receiving NCP handed it over to the destination user. If each connection were encrypted with a separate key, then NCP errors and misdelivery within the host operating system would not affect security. If suitable error correction is incorporated with the encryption, then integrity problems can also be detected.

The main problem in this approach is the initial connection establishment protocol—how to permit users to supply the NCP with parameters telling which site and what type of connection should be established, without large confinement channels in the system. For a discussion of these and related issues, see Kline.[4] The method of solution outlined there has been implemented in UCLA Unix. The additional kernel code to support secure network operation was quite small. Further, most of the original NCP was kept unmodified, although its lower level was altered to match the kernel interface.***

PROGRAMMING LANGUAGE ISSUES

The programming language employed in software development is usually recognized to have a significant effect on that effort; however when the goal of development includes verification, the effect is heightened. The specific language issues break down here into two groups—those concerned with systems programming, and those concerned with the scale of the verification steps.

Systems programming issues arise in the same way that they occur in most high-level systems programming languages. It is necessary to be able to express details of the hardware in the high-level language, such as interrupt vectors, hardware device registers, or special instructions. These facilities must be available in the programming language, but in a way that minimizes the effect on the semantics of the rest of the language.

Virtually all the security and integrity relevant code in UCLA Unix is written in a slightly altered Pascal. Obvious verification problems were removed from the language, such as pointers, variant records and various sources of aliasing.[5] I/O facilities were also deleted, since we were building I/O mechanisms, among other functions. The run-time package needed to support Pascal I/O would have been useless baggage, and since it typically would be written in assembly code there would be little chance of ever verifying properties of its operation.

It was also necessary however to add features to Pascal to permit systems programming, as remarked above. Very few additions were actually necessary, and were limited to the following:

1. The ability to declare a variable to be stored at a fixed physical location (to initialize interrupt vectors, access device control registers, etc).
2. Assembly language procedures (so that special hardware instructions could be expressed as a procedure call).

* Context switches on the PDP-11 are in general fairly slow. Therefore, the scheduler is to be changed so that it is not invoked at every process switch, but instead periodically gives the kernel advice about which processes are to be run. In this way, most of the scheduling algorithms remain out of the kernel, but the additional overhead of having two context switches per (desired) context switch is eliminated. That work was not done when this paper was authored.

** The NCP being considered was developed at the University of Illinois.

*** The Illinois NCP "kernel" was rewritten.

3. The ability to have procedures which take array parameters whose length is determined at call time (to remedy the most significant limitation of Pascal).

We also developed an extensive library system to support independent compilation of program modules, and yet force type integrity across module boundaries. The compiler and library system force recompilation of modules when needed for compatibility with another module which has been altered. This facility is needed since the verification work depends on type enforcement. The language, compiler, and library system are discussed by Walton.[13]

There are many issues concerned with the scale of the verification effort. It is believed that over half of the original verification effort could be avoided if the language contained more reasonable controls over aspects of program behavior. One of the more obvious examples concerns the integrity of global variables. An important portion of the assertions to be verified state that most of the kernel variables have not been altered by the routine being considered. (After all, much of the statement of security concerns what is not to happen.) These assertions, in the form of a large invariant, could be simply handled by scope controls in the language, such as the Import/Export lists of Euclid.[5] Then compile time enforcement could be employed and the verification task correspondingly simplified. UCLA Pascal has been modified to provide Import Lists.

Another example where the verification task can be eased concerns array bounds checking. In Pascal, many subscripts can easily be out of range, and therefore potentially reference data other than the given array, violating type rules. There are four reasonable ways to deal with this problem—subscript checking could be done by hardware, by runtime software generated by the compiler, by runtime software explicitly inserted by the programmer, or it could be verified in many cases that subscripts do not get out of range. The PDP-11 hardware base does not provide any reasonable way to itself check subscript references.† The UCLA Pascal compiler does not implement array checking code. Therefore, a combination of the remaining choices were taken. The resulting assertions which need to be proven compose a significant fraction of the total verification to be done. Clearly here is a fertile area for language support or enhanced verification tools.

ARCHITECTURAL OBSERVATIONS

UCLA Unix comprises the first verifiably secure, full functionality operating system with a fine grain of protection. The experience gained in its design and development led us to several conclusions. Most obvious, secure operating systems are feasible to develop, although the development cost is likely to be considerably greater than if highly reliable security and integrity were not such a serious goal. However, the result is a system which appears to exhibit considerably enhanced reliability and integrity, and because of the strict modularity, is easier to modify. Performance does not appear to be seriously affected by the architectural constraints imposed by the various goals. That is, the net result of the security goal seems to be a better system in general.

It should be noted, however, that one of the central ideas to the success of the work, kernel-structured architectures, requires considerable rethinking of the usual operating system architecture views if it is to be effectively employed. Much of the standard operating system wisdoms must be reexamined, or the result will be a "kernel" that is in fact overly complex and not suitable for a rigorous demonstration of correct security and integrity enforcement.

In conclusion, it appears that the goal of obtaining secure operating systems, at least for centralized, medium scale machines, has been largely reduced to (high quality) engineering, with the most significant progress required in program verification.

† The new, upward compatible DEC VAX/780 does.

BIBLIOGRAPHY

1. Fabry, R., "Capability Based Addressing," *Communications of the ACM*, Vol. 17, No. 7, July 1974, pp. 403-412.
2. Gaines, R. S., Private communication, 1977.
3. Kemmerer, R., "Verification of the UCLA Security Kernel: Abstract Model, Mapping, Theorem Generation and Proof," PhD Thesis, UCLA Computer Science Department, 1978.
4. Kline, C. S., "Protection Mechanisms for Operating Systems and Networks," PhD Thesis, UCLA Computer Science Department, 1979 (forthcoming).
5. Lampson, B. et. al., "Report on the Programming Language Euclid," *SIGPLAN Notices*, Vol. 12, No. 2, February 1977.
6. Millen, J., "Security Kernel Validation in Practice," *Communications of the ACM*, Vol. 19, No. 5, May 1976, pp. 243-250.
7. Organick, E., "The Multics System, an Examination of its Structure," MIT Press, 1971.
8. Popek, G., and D. Farber, "A Model for Verification of Data Security in Operating Systems," *Communications of the ACM*, September 1978.
9. Richie, D., and K. Thompson, "The Unix Timesharing System," *Communications of the ACM*, Vol. 17, No. 7, July 1974, pp. 365-375.
10. Saltzer, J., and M. Schroeder, "The Protection of Information in Computer Systems," *proceedings of the IEEE*, 1975.
11. Schroeder, M., D. Clark, J. Saltzer, "The Multics Kernel Design," *Proceedings of the Sixth Symposium on Operating Systems Principles*, W. Lafayette, Indiana, November 1977.
12. Walker, B., "Verification of the UCLA Security Kernel: Data Defined Specifications," Masters Thesis, UCLA Computer Science Dept., November 1977.
13. Walton, E., "The UCLA Pascal Translation System," UCLA Computer Science Dept. Technical Report, January 1976.
14. Urban, M., "A Policy Manager for UCLA Secure Unix," Masters Thesis, UCLA Computer Science Dept., 1979 (forthcoming).