ENGINEERING A SECURITY KERNEL FOR MULTICS

Michael D. Schroeder

September 23, 1975

This is a preprint of a paper to be presented at the ACM 5th Symposium on Operating Systems Principles, University of Texas, Austin, Texas November, 1975.

ENGINEERING A SECURITY KERNEL FOR MULTICS

Michael D. Schroeder

Massachusetts Institute of Technology
Project MAC and Department of Electrical Engineering and Computer Science
Cambridge, Massachusetts 02139

ABSTRACT

This paper describes a research project to engineer a security kernel for Multics, a general-purpose, remotely accessed, multiuser computer system. The goals are to identify the minimum mechanism that must be correct to guarantee computer enforcement of desired constraints on information access, to simplify the structure of that minimum mechanism to make verification of correctness by auditing possible, and to demonstrate by test implementation that the security kernel so developed is capable of supporting the functionality of Multics completely and efficiently. The paper presents the overall viewpoint and plan for the project and discusses initial strategies being employed to define and structure the security kernel.

Key Words: protection, security, privacy, security kernel, operating system structure, verification; Multics.

CR Categories: 4.3

Introduction

This paper describes a research project of the Computer Systems Research Division of Project MAC at M.I.T. The objective of the project is to engineer a security kernel for Multics. This project is part of an effort to develop a secure version of Multics that implements the information release constraints of the military security system. Involved in that effort are the Electronic Systems Division of the United States Air Force, the MITRE Corporation, and Honeywell Information Systems Inc.

The paper starts with a brief introduction to the computer security problem and the role that verification of correctness plays in producing a secure system. Next, the plan is outlined for evolving Multics into a system, based on a security kernel, whose security properties are easier to verify. The third section presents a general discussion of a security kernel as the structural basis for a secure system. Finally, to make the nature of the Multics kernel project more precise, a sample of specific activities underway or planned is presented.

The research reported in this paper was supported in part by Honeywell Information Systems Inc.; and in part by the Air Force Information Systems Technology Applications Office (ISTAO) and by the Advanced Research Project Agency (ARPA) of the Department of Defense under ARPA order No. 2641 which was monitored by ISTAO under contract No. F19623-74-C-0193.

The issue of security arises when a single computer system provides computation and information storage service to a community of users. As the functional advantages of such shared systems have been recognized, so has the need to include facilities for controlling the access of the various users to the contained information. Many systems now include protection mechanisms providing enforcement of specific patterns of externally specified constraints on the access of executing programs to the contained information [1]. For applications involving sensitive information, the usefulness of a shared system can depend upon the ability of its protection mechanisms to prevent unauthorized release, modification, and sometimes denial of use of the information it contains.

A system is secure if it is known to prevent all actions defined as unauthorized by the specification of its security properties. Penetration exercises involving several different systems have made it apparent that existing shared, general-purpose systems are not secure. In all such systems confronted, a wily user can construct a program that can defeat the access constraints supposedly enforced by the system. Design and implementation flaws exist that provide paths by which the protection mechanisms in the system can be circumvented, thus violating the security of the contained information.

Building a secure system is hard because security places negative requirements on a system. To be secure all possible ways to perform unauthorized actions must be blocked; no way to

circumvent the protection mechanisms can exist. A single flaw in the design or implementation is sufficient to allow a violation of security, and the absence of such flaws cannot be demonstrated by testing. For a system to be considered secure, a convincing logical verification that the implemented system is a correct realization of its security specification is required.

The operating systems of shared, general-purpose computers have a well-known tendency to be extraordinarily large and complex. This size and complexity interacts badly with the negative nature of security requirements. It generates many possible ways to perform unauthorized actions, some of which will go unnoticed by system designers and therefore remain unblocked by the protection mechanisms provided, and increases the probability of exploitable errors in the implementation of the protection mechanisms that are provided. It also makes the required verification of correctness impossible to perform. Available formal program verification techniques are overwhelmed and even verification by manual auditing is thwarted by the inability of one person to comprehend in detail the relevant software.

The negative nature of security properties is an intrinsic problem. The size and complexity of existing, shared, general-purpose systems, however, is not intrinsic. This research project attacks the problem of producing a secure system by exploring ways to reduce the size and complexity of the software that must be correct for claimed constraints on access to information to be enforced.

Method of Attack

The problem of constructing a secure system has attracted considerable interest recently and is being attacked with a variety of different strategies [2]. One approach being explored involves constructing a formal specification for the desired security (and other) properties of a system, and then, through a methodical, top-down design and implementation process, building a matching, operational system. The correspondence of the implemented system with the formal security specification is to be proved using formal program verification techniques. The hope is to achieve a level of confidence in the match of the system to the specification similiar to the confidence that a mathematician has in the result of a well-wrought proof. A step necessarily left to human intuition is determining how appropriate the security properties expressed in the formal specification are to a particular real-world application. Three current projects [3,4,5] are trying to produce simple, experimental examples of secure systems using different versions of this approach.

At the other end of the spectrum are efforts [6,7] to find, catalog, and repair security flaws in existing systems. The goals are to convince skeptics that the computer security problem is real, to understand the sort of flaws that can be exploited, and to try to reduce the ease with which available systems can be penetrated.

Our project to reduce the size and complexity of the software that must be verified correct to produce a secure system falls somewhere between these two extremes. Our plan is to evolve Multics [8], a general-purpose, remotely accessed, multiuser system, into a prototype system with the same essential features, but with a small and simple protected central core of software. This core will be a security kernel embodying all mechanisms necessary to enforce the claimed constraints on access to information in the system. The goal is a kernel sufficiently small, well-structured, and easy to understand that feasibly an expert could verify its correctness through manual auditing. Such a kernel also may be susceptible to verification through formal techniques, although the programs may have to be rewritten in a more appropriate style. The kernel will enforce access constraints that combine nondiscretionary controls reflecting the information release policies of the military security system and the discretionary controls on information sharing that were part of the original Multics design. (1)

The choice to evolve an existing system rather than produce a new one follows from our interest in problems of system structure, rather than in techniques for formal specification of security properties, structured programming, or formal verification of program correctness. Our goal is developing a security kernel that can be demonstrated to support the full set of functions that are desirable in a shared, general-purpose system. With the current understanding of computer systems, it is hard to have confidence that the full implications of a system structure are understood without complete implementation. Designing a new system without building it or building a simple, experimental system (2) would not allow the completeness of our kernel design to be tested adequately. Thus, to start fresh would mean undertaking the entire job of building a new, general-purpose system. By developing a security kernel for an existing, general-purpose system, we avoid this enormous effort.

A combination of factors makes Multics well suited as a base from which to engineer a security kernel. To start with, Multics provides a full set of functional capabilities, including high-bandwidth direct sharing of information among computations [11]. In addition, Multics has been developed from the ground up to protect the information it contains from unauthorized access. It already includes general protection mechanisms to control information sharing among users [10] and provides direct hardware support for some of these mechanisms [12]. Minimal features to support the information release constraints of the military security system recently have been added [13]. Thus, the system exhibits a set of security

---

(1) A formal specification [9] of the nondiscretionary controls is being developed by a group at MITRE. An informal specification of the discretionary controls is available in [10].

(2) Two examples of features usually left out of experimental systems that can complicate the kernel of an complete, general-purpose system are the storage quota and backup mechanisms mentioned in a later section.

properties that would be interesting to implement correctly and provides hardware support that will make the job easier. Also, the system is well organized for evolution and modification because it is relatively modular, is largely written in PL/I [14], and was originally constructed with evolution as a primary objective. Finally, Multics provides two unique opportunities to test and export the results. First, because Multics is a commercially available product, ideas developed in the course of this research that simplify the system's structure without changing its functionality or reducing its efficiency can be added to the standard system. Second, the security kernel being produced is serving as the structural basis for the secure version of Multics being developed by the Air Force, MITRE, and Honeywell in an effort of which our project is part.

The Security Kernel

A security kernel, a minimal, protected core of software whose correct operation is sufficient to guarantee enforcement of the claimed constraints on access, is the structural basis for organizing a secure system. Rather than being dispersed throughout the system software, all protection mechanisms are collected in the kernel, so that only this kernel need be considered in order to verify that the specified security properties are implemented correctly.

The security specification that a particular system must match limits how small and simple (3) the kernel can be. The patterns of access constraints to be enforced is an obvious factor. The set of abstract objects and operations to be controlled may be even more important. In Multics, it appears that most of the mechanism in the kernel will implement the abstract objects protected by the system, for example, processes, segments, directories, and I/O streams, and will be relatively independent of the specific patterns of access constraints enforced by the system.

A characterization of the mechanisms that should be included in a security kernel can be obtained by viewing the security specification as a set of constraints on the interaction of the various computations that occur in a computer system. The protection mechanisms of the system prevent one computation from exerting an unauthorized influence on the input, progress, or output of another. Permanently stored data is one form of the input and output of computations. This view suggests that the security kernel should embody all system-provided mechanisms that are common to more than one computation (domain), because a common mechanism is required if one computation is to influence another. A mechanism is common to two computations if it uses some set of data items whose value one computation can influence and the other can notice. The influence and notice may be direct--one writes into a data item and the other reads it--or indirect--the invocation of a procedure by one somehow alters the procedure's internal state so that the outcome of an invocation by the other is affected. Common mechanisms are required to implement any explicit or implicit communication among computations. Thus, mechanisms implementing information sharing, interprocess communication, and physical resource multiplexing must be common. If no communication is involved, however, then a common mechanism is not required to implement a function. Common mechanisms carry a built-in risk--they make it possible for the computation of one user to exert unauthorized influence over the computations or data of another. Malicious users must exploit flaws in common mechanisms to work their will. To thwart such malicious activity the system designers must ensure that the common mechanisms have no exploitable design or implementation flaws, and must protect the common mechanisms against tampering. Thus, a security kernel should be the least amount of common mechanism necessary to implement the patterns of information sharing, interprocess communication, and physical resource multiplexing that are desired in the system. (4)

Although a security kernel contains all the mechanisms that must be verified as correct to produce a secure system, a correct kernel does not guarantee the integrity of all computations or stored data in a system. Nonkernel software still can cause undesired release of information, modification of information, or denial of its use. But if the kernel is correct, then these undesired results will not be unauthorized. To understand the meaning of this distinction, consider the nonkernel software as grouped in four categories.

First, there are the system-provided programs that execute as part of user computations. These include the library subroutines available in most systems and all the programs usually part of a supervisor that are not included in a security kernel. These system-provided programs are not common mechanisms, even though in many systems all computations may share the same nonwriteable code that embodies their algorithms. This is so because a private copy of the alterable part of these programs, the variable data, is provided for each computation. Because they are private mechanisms, no interuser interaction can occur through them. Private mechanisms may contain errors, but these errors can by triggered only by the actions of the computation that they might damage as a result. If one assumes that the system programmers who constructed them are not malicious and did not willfully plant "trojan horses," then the mistakes caused by these system-provided programs will decrease in time as all normally used functions are exercised. Under these circumstances the threat posed by a potential random error causing undesired release, modification, or denial of a user's data is acceptable for most applications. Unlike the common mechanisms of the security kernel, the nonkernel system-provided programs are not

---

(3) Unfortunately, no objective measure of overall complexity is known. The degree of complexity must be estimated subjectively.

(4) According to this characterization of a security kernel, a usual reason for including a mechanism in a supervisor, to protect it from accidential breakage caused by errors in user code, is not in itself sufficient to include a mechanism in a security kernel. Nonkernel mechanisms can be protected by placing them in other domains that are private to each user's computation.

susceptible to willful exploitation by other users. In any case, a user unsatisfied with their trustworthiness may choose not to use them and substitute his own programs.

The second category of nonkernel software is programs constructed by a user and executed in that user's computations. Any undesired result caused by errors in these is the user's own problem. The only possible help to the user would be providing tools to aid verifying the correctness of his own programs.

The third category, possible in many systems, is programs borrowed from other users. These are a real danger to the borrower's computations. Borrowed programs can contain trojan horse code maliciously constructed to cause results undesired by the borrower. (5) A user should borrow programs from another only when the borrower has reason to trust the lender. The inclusion of security kernel facilities to support user-constructed protected subsystems can reduce the potential damage such a borrowed trojan horse can do by isolating it in a separate domain of the borrower's computation. Due to the confinement problem [15], however, and also to the possibility of a borrowed program disrupting the borrower's computation simply by calculating incorrect results, a user-initiated verification of the borrowed program is the only complete protection.

The fourth category is common mechanisms that a group of users sets up to implement some function involving interuser communication or coordination. For example, a team producing a new compiler might set up a program development subsystem with a common mechanism to control installation of new modules into the evolving compiler. Such a mechanism makes the group susceptible to undesired interaction in the same way that an insecure supervisor does for the whole user community. If a user agrees to become party to such a common mechanism, then he must satisfy himself of its trustworthiness.

From considering these four categories of nonkernel software it is apparent that the essential mechanisms to verify correct are the common mechanism of the security kernel. The security kernel has initial control of all paths for the interaction of computations and every user of the system is forced to rely upon it. A correct kernel provides the tools with which a user may protect his computations and data against unwanted interference from the computations of other users. In a system providing for direct sharing of programs and data, however, users can agree to cooperate in ways that the security kernel cannot control. The kernel can prevent such sharing unless it is explicitly authorized, but cannot completely control the interaction between user's that agree to share. The security kernel prevents activities that the security specification for the system defines as unauthorized, but not all undesired results are caused by unauthorized activities.

A fifth category of nonkernel software also needs to be considered. One important technique for simplifying the structure of the security kernel is writing it with a high-level programming language. Using a high-level language to generate the kernel seems to require that the compiler also be verified correct, a troubling thought since the compiler may well be larger than the kernel. Verification of correct function may be less of a problem for the compiler, however, than for the kernel. The kernel must work correctly for all possible inputs; the compiler must compile correctly only the specific programs of the kernel--not all possible programs. Thus, the compiler's effect on the kernel can be determined by comparing the source code specifications for each kernel module with the compiler-produced object code implementation, a task much simpler than verifying the compiler correct for all possible source programs. (6)

A Kernel for Multics

Engineering a security kernel for a system requires isolating a minimum set of functions capable of supporting the system and finding a way to structure the kernel to facilitate verifying its correctness. Our plan is to produce a security kernel for Multics by removing nonkernel mechanisms from the supervisor, and restructuring the remaining kernel and partitioning it into multiple protection domains. This section describes these three interrelated categories of activities and provides specific examples of work underway or planned in each. The intention of the section is to communicate the spirit of the work rather than to to discuss thoroughly the various activities. The detailed results of individual activities are being communicated in other reports [16,17,18].

The first category of activities is taking functions not requiring implementation as common mechanisms out of the supervisor to be implemented in the user domains of a process. In many cases this transfer involves undoing a pattern caused by a performance characteristic of the original Multics implementation for the Honeywell 645 computer. For that machine, the multiple protection domains of a process, the so-called protection rings, were simulated in software. Cross-ring calls were quite expensive: a call that went from a user ring in a process to the supervisor ring cost much more than a call that did not change protection rings. The effect on system structure can be seen by considering two procedures, A and B. If a single invocation of A can result in a flurry of calls from A to B, then there is a performance benefit in placing both A and B in the supervisor, even if only B needs to be part of the protected, common mechanism. As a result of this performance characteristic of the

(5) This is a special case of a common mechanism. The data item whose value the lender can cause to change (and thereby influence the computation of the borrower) is the code of the borrowed program itself. Even if the program is nonwritable when borrowed, it was written by the lender when constructed.

(6) Use of a high-level language for kernel construction can generate a dilemma. An optimizing compiler can increase system efficiency, but may make impossible matching object code with source code to verify correct compiler function.

645 implementation, many functions that did not need to be implemented as common mechanism were included in the supervisor.

The present hardware base for Multics, the Honeywell Level 68, implements protection rings in hardware. Calls from one ring to another cost no more than calls inside a ring. With the performance penalty associated with supervisor calls removed, many modules included in the supervisor for performance reasons rather than protection reasons now can be removed. (7)

Actually, removing a module from the supervisor is more difficult than the example suggests. In most cases, the common and private parts of a facility are not neatly packaged in separate procedures but are intricately intertwined in the same supervisor procedures and data bases. The key problem is decomposing the supervisor into common and private primitive functions.

Most removal activities have centered on the file system. In one activity, now completed, the functions of dynamic intersegment linking and direction of file system searches to satisfy symbolic references have been removed from the supervisor [16,17]. This removed a vulnerable and complex mechanism from the supervisor. The vulnerability is a result of the linker having to accept user-constructed code segments as input data. Numerous accidents had shown that such a complex "argument", if maliciously misstructured, could cause the linker to malfunction while executing in the supervisor. Removing the linker eliminated 10% of the gate entry points into the supervisor and 6% of the total object code. The linker's removal also demonstrated that linking procedures together across protection boundaries (rings) could be done without resorting to a mechanism common to both domains.

A second completed activity removed from the supervisor the facilities for managing the association between reference names and the segments in the address space of a process [18]. Taking this naming mechanism out of the supervisor required that a data base central to the management of the address space, the Known Segment Table, be split into a private part that maintains the binding between reference names and segment numbers and a common part that maintains the binding between segment numbers and segments. Removal reduced fivefold the size of the protected code needed to manage the address space of a process. It also provided a new, simpler interface to the file system portion of the supervisor. Instead of identifying a directory by character string tree name locating it in the file system hierarchy, a segment number now is used. The algorithms for following a tree name through the file system hierarchy to locate the named element are now implemented by procedures executing in the user ring. (The actual file system hierarchy remains protected inside the security kernel.) Because tree names are now resolved one component at a time, the kernel had to learn to lie on occasion about the existence of file system directories. This deception keeps the kernel from divulging a directory's existence before an accessible element in the subtree rooted by the directory is found.

An activity under investigation involves making most of system initialization execute once, in a user environment of a previous system version, instead of executing inside the supervisor each time the system is started. The change is to produce a system tape with a bit pattern that, when loaded into memory, manifests a fully initialized system. At present the system bootstraps itself in a complex way each time it is loaded from a tape containing the separate pieces. The new pattern of operation removes most initialization software from the kernel. The correct initialization of the kernel also should be easier to verify, for most of the work of initialization will occur when the tape is made, in the stable environment of a fully initialized system.

Another activity is exploiting the equivalence between entering a protected subsystem and creating a new process in response to a user's log-in. The goal is to make a single mechanism do both tasks, so that the privileged, protected code used to authenticate and log-in users can be executed in the user code protection environment. The authentication algorithm still must be verified not to malfunction if the user trying to log-in behaves unexpectedly. Such verification should be easy, however, since the user/system interface severely limits user behavior.

The second category of activities is restructuring mechanisms that must remain in the kernel. Such activities can reduce both the size and the complexity of the kernel. In some cases a piece of the kernel can be eliminated and its function assumed by another kernel mechanism. For example, one activity is exploring replacement of all external I/O mechanisms (to terminals, tape drives, card readers, card punches, and printers) with the ARPA Network attachment. This would eliminate many special mechanisms for managing I/O devices and leave a single mechanism for managing the network attachment. Internal I/O functions (for managing the virtual memory, performing backup, and loading the system) would still be managed in the kernel.

A proposed buffering strategy for network input uses the virtual memory to provide a core resident buffer that appears to be infinite in length. With the present circular buffer, which has to be used over and over, complex mechanisms are required to cope with messages not removed before a complete circuit of the buffer is made. The circular buffer scheme is really providing a special-purpose storage management facility. The proposed infinite buffer uses instead the standard storage management facility of the system--the virtual memory.

Several restructuring activities involve the implementation and use of processes. One activity, now nearing the end of the design phase, is a reimplementation of processes using two layers of

---

(7) There may still exist other performance penalties associated with removing functions from the supervisor that will inhibit production of the smallest possible kernel. One goal of the research is to understand better the performance cost of security.

mechanism. (8) This new design simplifies the interaction of the process implementation with the virtual memory management machanisms. It also simplifies the base-level interprocess communication mechanisms of the system. The first level of mechanism multiplexes the processors into a larger fixed number of virtual processors. Because the number of virtual processes is fixed, this layer need not depend on the mechanisms for managing the virtual memory. Several of the virtual processors are permanently assigned to implement processes for the dedicated use of other kernel mechanisms, including the virtual memory management mechanism. The remaining virtual processors are multiplexed by the second layer of the process implementation into any desired number of full Multics processes that execute in the virtual memory. Use of the proposed base-level interprocess communication facility can be controlled with the standard memory protection mechanisms.

The implementation (implied above) of certain kernel mechanisms as asynchronous parallel processes also simplifies system structure, which now forces many supervisor mechanisms into sequential algorithms. The virtual memory mechanisms for moving pages among the three levels of the memory hierarchy are a good example. Whenever a missing page fault occurs in a process, the fault handler attempts to initiate the transfer of the desired page from bulk store or disk to primary memory. This can be done only if a free primary memory block is available. If none is available the fault handler must move a page from primary memory to the bulk store to make room. This, in turn, is possible only if a free block of bulk store is available. If not, a page must be moved from the bulk store, via primary memory, to a disk. At present, this series of steps occurs sequentially with page control executing in the process that took the page fault, and then in various other user processes that happen to receive the subsequent I/O interrupts. The new scheme involving multiple dedicated processes is much simpler. One process makes sure that some small number of free primary memory blocks always exist. Whenever the number of free primary memory blocks drops below that number, this process transfers pages to bulk store. Another process keeps space free on the bulk store by moving pages to disk when required. Signals from processes that have taken a page fault and not found free primary memory blocks activate the process that frees primary memory. The process that frees bulk store blocks is driven in a similar manner by signals from the process that frees primary memory blocks. The path taken by a user process on a page fault is greatly simplified. This process can just wait until a primary memory block is free and then initiate the transfer of the desired page into primary memory.

Interrupt handling is another possible application of processes in the kernel. Each interrupt handler would be assigned its own process in which to execute, rather than being forced to inhabit whatever user process was running when the interrupt occurred. As a result, the system interrupt interceptor could turn each interrupt into a signal to the corresponding process. Being processes, the interrupt handlers could use the normal system interprocess communication mechanisms to coordinate their activities with one another and user processes, greatly simplifying their structure. The problem to solve here is implementing the interrupt processes so that system performance is not degraded.

A major activity is restructuring the portion of the file system that must remain in the kernel. This software implements the directory hierarchy and manages the virtual memory at the level of segments. Work in this area is just beginning, but three changes with a potential, significant cumulative effect are promising first steps. One change is removing physical attributes of segments from directory entries. The physical attributes will be recorded in data bases associated with the various secondary storage devices. (This modification is being implemented as part of a file system overhaul done by Honeywell for other reasons.) The second change is removing storage quotas from directory entries, recording them instead in a separate (possibly hierarchical) data structure. The third is eliminating the dependence of the backup mechanism on the date-time modified information recorded in directory entries and reflected up the hierarchy toward the root. Eliminating this dependence will allow the date-time modified item to be removed from directory entries as well. Backup will be driven by a queue of requests from the machinery that controls deactivation of segments from primary memory. As a result of these three changes, it appears that the management strategy for the Active Segment Table can be modified to eliminate the need for holding active the superior directories of an active segment [11].

The various restructuring activities eventually will extend to all parts of the kernel, and to its overall structure.

The third category of activities is partitioning the kernel into differently protected pieces to modularize the job of matching the kernel to the system security specification. (9) The specific projects in this category are not as well developed as the others. There appear to be several different design principles with which to generate the kernel partitions, and it is not yet clear which produces the kernel that is easier to verify. To illustrate two possible approaches to partitioning a kernel into multiple domains, imagine that the security specification is expressed as a set of security properties, each of which must be met. One design principle is to divide the kernel into domains arranged so that each property is implied by a subset of the domains. Then, to verify that the kernel implements the security specification, an independent verification of each property is required, but each involves only a subset of the

(8) This idea is being explored by others as well [3,19].

(9) Partitioning is really the same problem as dividing the kernel into separate procedures and data bases, with the extra property that the modularity is enforced by the system's protection mechanisms

domains in the kernel. Another design principle is to ignore any structure suggested by the security properties and divide the kernel into domains according to some other principle of structured programming (for example, Parnas' notion of information hiding [20]) so that each domain has a simple interface behavior specification. Verification of each of the security properties may involve all the kernel domains, but once each domain has been verified to match its interface specification, then only these specifications need be considered to verify each security property. Which of these two approaches is preferable--or indeed, whether they really are different approaches--remains to be seen.

Two specific methods for partitioning the Multics kernel are available. The first is dividing the part of the kernel that is in the address space of each process into multiple layers in different rings of protection. The second is placing some of the kernel processes in separate address spaces and also using the protection rings to layer them. Several suggestions have been made for layering the part of the kernel that is in each user process. One is that the bottom layer implement a file system in which all segments were named by system-generated unique identifiers. The next layer would implement a naming hierarchy on top of the primitive first-layer file system. Another suggestion is that mechanisms to provide the nondiscretionary controls on the flow of information among processes be implemented at the bottom and mechanisms to control discretionary sharing within the constraints of the nondiscretionary controls be implemented in the next layer. This last suggestion is particularly intriguing, because if correctly done, the notion of minimizing common mechanisms would be well supported. The second-layer mechanisms would be common only within the access constraints enforced by the first layer.

Partitioning through use of separate address spaces for kernel processes is being considered in the case of the processes that manage the various system resources. The protection rings in these processes then could be used to separate the policy and mechanism components of the resource managers. (10) For example, the process described earlier that removed pages from primary memory could be given its own address space with multiple rings. Programs in the most privileged ring would implement the mechanics of page removal, providing gate entry points for requesting the movement of a particular page from primary memory to a particular free block on the bulk store, and for obtaining usage information about pages in primary memory. The policy algorithm that decides which page to remove when another free primary memory block needs to be generated would execute in a less privileged ring, calling the gate entry points to collect the necessary usage statistics and to do the actual moving, once a decision was made. The policy algorithm, however, could never read or write the contents of pages, learn the segment to which each page belonged, or cause one page to overwrite another. Such operations would not be available in its ring of execution. The result is that the policy algorithm could never cause unauthorized use or modification of the information stored in the pages. It could only cause denial of use. Under the circumstance that denial of use was deemed less serious than the other security violations, the policy algorithm need not be as carefully verified as the rest of the kernel. It appears that the idea of separating policy from mechanisms applies to all resource management processes.

(10) Separation of policy from mechanism is a structural principle that has been explored by many others [21,22,23].

Conclusion

This paper has presented the plan for a research project to evolve the Multics supervisor into a security kernel capable of supporting the functionality of Multics completely and efficiently. It has described a sample of the specific strategies being employed. The broad objective is finding ways to reduce the size and complexity of the software that must be correct for a shared general-purpose system to be secure. Reduced size and complexity of security-relevant software is a prerequisite to performing a convincing logical verification that a system correctly implements the claimed access constraints, no matter what verification techniques are used. Without such verification of correctness, a system cannot be considered secure.

At the time this paper is being written, the project has run for about half of its intended four year span, and most of the initial tasks are nearing completion. So far, the expected reductions in size and simplifications of structure of the security-relevant software seem to be occuring. It is too soon to tell, however, whether the security kernel for Multics that will result will be sufficiently small and simple to be understood in detail by one person.

Acknowledgements

In describing a group project of the Computer Systems Research Division of Project MAC at M.I.T., this paper discusses the work of several faculty members, graduate students, and staff members in the Division. Rather than list all here, they will receive credit for their contribution as specific activities are completed and reported separately. Preparation of this paper was aided by written commentaries on various drafts provided by E. Cohen, F. Corbató, R. Fabry, D. Hunt, P. Janson, D. Reed, J. Saltzer, and R. Schell, and by comments from D. Clark, A. Jones, and D. Redell.

References

[1] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," Proc. IEEE 63, 9 (Sept. 1975), pp. 1278-1308.

[2] J. H. Saltzer, "Ongoing Research and Development on Information Protection," ACM Operating Sys. Review 8, 3 (July 1974), pp. 8-24.

[3] L. Robinson, et. al., "On Attaining Reliable Software for a Secure Operating System," Int. Conf. on Reliable Software, Apr. 1975, pp. 267-284.

[4] G. J. Popek and C. S. Kline, "A Verifiable Protection System," Int. Conf. on Reliable Software, Apr. 1975, pp. 294-304.

[5] W. L. Schiller, "Design of a Security Kernel for the PDP-11/45," The MITRE Corp. Tech. Rep. ESD-TR-73-294, Dec. 1973.

[6] J. Carlstedt, R. L. Bisbey II, and G. J. Popek, "Pattern-Directed Protection Evaluation," U. of So. Calif. Inf. Sci. Institute Tech. Rep. ISI/RR-75-31, June 1975.

[7] P. A. Karger and R. R. Schell, "Multics Security Evaluation: Vulnerability Analysis," Air Force Elec. Sys. Div. Tech. Rep. ESD-TR-74-193, Vol. II, June 1974.

[8] F. J. Corbató, J. H. Saltzer, and C. T. Clingen, "Multics - the First Seven Years," AFIPS Conf. Proc. 40 (SJCC 1972), pp. 571-583.

[9] D. E. Bell and L. J. LaPadula, "Secure Computer Systems," The MITRE Corp. Tech. Rep. ESD-TR-73-278, Nov. 1973.

[10] J. H. Saltzer, "Protection and the Control of Information Sharing in Multics," CACM 17, 7 (July 1974), pp. 388-402.

[11] A. Bensoussan, C. T. Clingen, and R. C. Daley, "The Multics Virtual Memory: Concepts and Design," CACM 15, 5 (May 1972), pp. 308-318.

[12] M. D. Schroeder and J. H. Saltzer, "A Hardware Architecture for Implementing Protection Rings," CACM 15, 3 (Mar. 1972), pp. 157-170.

[13] Honeywell Information Systems Inc., "Design for Multics Security Enhancements," Air Force Elec. Sys. Div. Tech. Rep. ESD-TR-74-176, 1974.

[14] F. J. Corbató, "PL/I as a Tool for System Programming," Datamation 15, 6 (May 1969), pp. 68-76.

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

[16] P. A. Janson, "Removing the Dynamic Linker from the Security Kernal of a Computer Utility," S.M. Thesis, Dept. of Elec. Eng. and Comp. Sci., M.I.T., June 1974. (Also available as Project MAC Tech. Rep. MAC-TR-132, June 1974.)

[17] P. A. Janson, "Dynamic Linking and Environment Initialization in a Multi-Domain Computation," ACM 5th Symp. on Operating Sys. Principles, Austin, Texas, Nov. 1975.

[18] R. G. Bratt, "Minimal Protected Naming Facilities for a Computer Utility," S.M. Thesis, Dept. of Elec. Eng. and Comp. Sci., M.I.T., July, 1975. (Also available as Project MAC Tech. Rep. MAC-TR-156, Sept. 1975.)

[19] A. R. Saxena and T. H. Bredt, "A Structured Specification of a Hierarchical Operating System," Int. Conf. on Reliable Software, Apr. 1975, pp. 310-318.

[20] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems Into Modules," CACM 15, 12 (Dec. 1972), pp. 1053-1058.

[21] M. J. Spier, T. N. Hastings, and D. N. Cutler, "An Experimental Implementation of the Kernel/Domain Architecture," ACM Operating Sys. Review 7, 4 (Oct. 1973), pp. 8-21.

[22] G. R. Andrews, "COPS - A Protection Mechanism for Computer Systems," Computer Sci. Teaching Lab., Univ, of Wash., Tech. Rep. 74-07-12, July 1974.

[23] W. Wulf, et. al., "HYDRA: The Kernel of a Multiprocessor Operating System," CACM 17, 6 (June 1974), pp. 337-345.