PROJECT MAC June 3, 1975

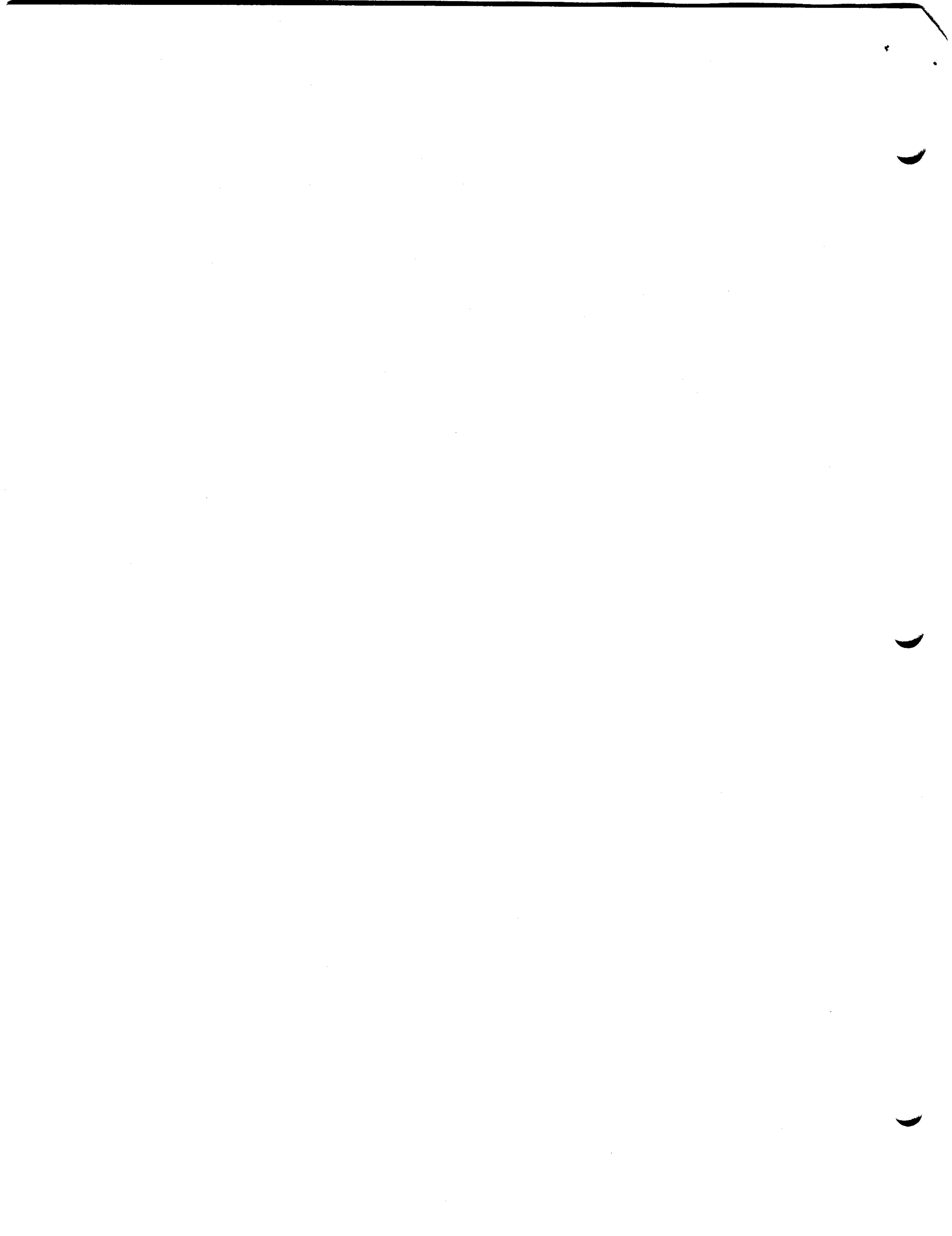
Computer Systems Research Division Request for Comments No. 80

ENGINEERING A SECURITY KERNEL FOR MULTICS

by Michael D. Schroeder

Attached is a draft of a paper describing the certification project that I am submitting to the 5th Symposium on Operating Systems Principles, to be held in November. This stuff is exceedingly hard to write down, and I would very much appreciate your comments and suggestions.

This note is an informal working paper of the Project MAC Computer Systems Research Division. It should not be reproduced without the author's permission, and it should not be referenced in other publications.



Engineering a Security Kernel for Multics

Michael D. Schroeder

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

DRAFT

Abstract

This paper describes a research project to engineer a security kernel for Multics, a general-purpose, remote-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 minimal mechanism to make certification by auditing possible, and to demonstrate by test implementation that the security kernel so developed is capable of supporting the complete functionality of Multics. The paper presents the overall viewpoint and plans for the project, and discusses the initial strategies being employed to isolate and structure a functionally adequate security kernel.

Key Words: protection, security, privacy, security kernel, certification, Multics.

CR Catagories: 4.3

The research reported in this paper was supported by ARPA, the Air Force, and Honeywell. (Appropriate contract numbers, etc., to be supplied.)

Introduction

This paper describes a research project of the Computer Systems Research Division of Project MAC at M.I.T. to engineer a security kernel for Multics. The objective of the project is to produce an example of a general-purpose, remote-accessed, multiuser computer system that can be certified, by auditing, to implement correctly the claimed constraints on the access of executing programs to the contained information. The paper presents the context, viewpoint, and plans for the project, and describes a sample of the initial strategies being employed to define and structure a security kernel for Multics.

The need for certified protection mechanisms arises when a single 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 for computer enforcement of intricate, externally specified policies for control of access by executing programs to the contained information [1]. The presence of such mechanisms, however, is not enough. Users, whether they be individuals or organizations, must have confidence in the integrity of the protection mechanisms before they can entrust sensitive data to a system. The system must be certified to implement without failure the desired control policies.

There are three ways in which the security of information

stored in a computer system can be violated:

1. Unauthorized release: an unauthorized person is able to discover the content of information stored in the computer.

2. Unauthorized modification: an unauthorized person is able to cause changes to stored information.

3. Unauthorized denial of use: an unauthorized person can prevent legitimate reference to or modification of stored information.

In a shared system the unauthorized person with respect to a specific act may be an otherwise legitimate user of the system.

In practice, producing a system that actually does prevent all such unauthorized activities has proved to be extremely difficult. Sophisticated users of most currently available systems are probably aware of at least one way to deny all users access to the stored information by deliberately crashing the system. Penetration exercises involving a large number of different systems have shown that, in all general-purpose systems confronted, a wily user can construct a program that can obtain unauthorized access to information stored within the system. (1)

The primary reason for these failures is the presence of design and implementation flaws that provide paths by which the access constraints supposedly enforced by the system can be circumvented. Underlying this cause are two interacting

---

(1) A recent paper by Linde [2] catalogs many of the penetration techniques that have been used.

difficulties. The first is that preventing all unauthorized acts is a negative kind of requirement. It is intrinsically quite hard to show that this requirement has been met, for one must demonstrate that no means for violating data security exist. The second is the well-known tendency for the operating systems of shared, general-purpose computers to be extraordinarily large and complex. This tendency interacts badly with the need to show non-existence of paths for violating data security, by providing a very complex environment in which to attempt the demonstration.

Certifying the protection facilities of a computer system has two aspects. One is determining that the patterns of access constraints claimed to be enforced by the system are appropriate to the intended application. The other is verifying that the system actually enforces the claimed constraints. The access constraints claimed to be enforced are expressed as a model defined in terms of the abstract objects and operations implemented by the system. Determining the adequacy of the match between such a model and the perceived requirements of the intended application requires human judgement. In this aspect defining different degrees of certification is difficult. On the other hand, the match between the model and the enforcement mechanisms of the system must be exact, for the model is expressed in terms of the objects and operations implemented by the system, and any difference represents a failure of the system to implement the claimed access constraints. In this aspect different degrees of certification can be defined, and reflect

different degrees of certainty that the match is exact.

Certification results in the certifier signing-off on a statement of adequacy. By signing, the certifier assumes responsibility for future security failures. A system is certifiable if the certifier can be convinced to sign.

One reason that current general-purpose systems are not considered certifiable is that available techniques, both formal and informal, for expressing their claimed security properties result in models so intricate that a certifier cannot determine the usefulness of the claimed access constraints. The other reason is that the systems themselves are so large and complex that a certifier has no way to develop confidence in the match between the actual protection mechanisms and any understandable model. The size and complexity of these systems thwarts verification of the match through manual auditing by making it impossible for one person to comprehend the entire mass of security-relevant software in detail, and overwhelms the available systematic program verification techniques. Making general-purpose computer systems certifiable requires developing better techniques for modeling the claimed security properties of systems and finding ways to reduce the size and complexity of the mechanisms that must be correct. This project addresses the latter requirement.

Method of Attack

The problem of constructing a certifiably secure system has attracted considerable interest recently and is being attacked with a variety of different strategies [3]. The key to the ultimate solution appears to be methodical design and construction techniques that systematically exclude flaws that can be exploited to produce security violations. Many imagine being able to construct a formal specification for a system, prove desired security (and other) properties about the specification, and then, by essentially mechanical steps, construct the implied operational system. The match between the model and the mechanism is a natural result of the systematic generation of the mechanism from the model. The hope is to achieve a level of confidence in the match of the mechanism to the model similiar to the confidence that a mathematician has in the result of a well-wrought proof. The only step of certification left to human judgement is determining the appropriatness of the properties expressed in the security model to a particular real-world application. To this end, many research groups are conducting investigations into methods of proving assertions about programs and program-like specifications, methods for formally describing security properties, and techniques of top-down program construction by successive refinement of descriptions of algorithms and data structures. At the other end of the spectrum, several groups are engaged in finding, cataloguing, and repairing security flaws in

existing systems with the aim of convincing skeptics that the problem is real, of understanding the sort of flaws that can be exploited, and of trying to reduce the ease with which the security of available systems can be violated. Somewhere between these two extremes are several groups [4,5], including our own, trying to produce examples of systems that can be certified.

Our approach is to attack directly the size and complexity exhibited by existing general-purpose systems. The plan is to evolve an existing, commercial, general-purpose computer system, Multics [6], into a prototype operating system with all the essential features of the present system, but with a small and simple protected central core. This core will be a security kernel embodying all mechanisms necessary to enforce the presumed security properties of the system. The goal is a kernel sufficiently small, well-structured, and easy to understand that certification through manual auditing by an expert is feasible. Such a kernel also may be susceptible to certification through more systematic program verification techniques. Isolation and simplification of this security kernel will be guided by an informal (but detailed) model of the presumed security properties of Multics coupled with a formal model (2) of a subset of these properties that is being developed by a group at the Mitre Corporation in a closely related effort.

---

(2) The formal model specifies a set of access constraints that restrict information flow in a hierarchy of compartments to patterns consistent with the national security classification scheme.

The choice to evolve an existing system rather than design a new one from scratch follows from a desire to demonstrate the ability of simplified mechanisms to support a complete, operational system providing the full set of functional capabilities that seem desirable in a general-purpose system. With the current state of understanding of computer systems, it is hard to have confidence that the full implications of a system structure are understood without complete implementation. If an operational system is not the goal, it is very easy to leave out many of the complexity-producing convenience features that the users demand of a production system. A structure which gracefully supports a toy system may be badly strained by the performance levels and extra features required in its real descendant. Also, the kind of design and implementation errors that produce security flaws in real systems tend to be less of a problem in toy systems. Thus, to start fresh would require undertaking the complete job of implementing a new general-purpose system in order to test the completeness of a kernel design. By developing a security kernel for an existing, operational system, the enormous effort of a complete, new system implementation is avoided.

A combination of factors makes Multics in particular seem well suited to serve as a base from which to engineer a certifiably secure system. To start with, Multics provides a full set of functional capabilities, including high bandwidth direct sharing of information among computations [7]. In

addition, Multics has been developed from the ground up to protect the information it contains from unauthorized access. It already includes protection mechanisms as advanced as any available [8], and provides direct hardware support for some of these mechanisms [9]. Thus, the system both exhibits a set of protection features that would be interesting to certify and provides protection features that will make the job of certification easier. Also, the system is better organized than most for evolution and modification, because it is relatively modular, is largely written in PL/I [10], and was originally constructed with evolution as a primary objective. Finally, because Multics is a commercially available product and new ideas developed in the course of this research should be relatively easy to retrofit to the standard system, the result, if successful, can be easily exported in a directly useful way.

Although the original design of Multics was very methodical, and the system is, if anything, already less complex in organization than most contemporary computer operating systems with similar functional goals, potentially, it could be supported with mechanisms that are much simpler yet. The intense pressure of initial implementation did not permit time for contemplation and development of simpler supporting structures. Included in the programs which perform essential supervisor functions are functions which could be done as well without the special powers and privileges of the supervisor. Most supervisor programs have been designed with primary attention paid to providing a broad

range of function and some have been designed with primary attention paid to efficiency; less attention has been paid to simplicity of organization and almost none to organizations that might be susceptible to systematic specification and verification. The basic premise of this research is that one wave of simplification applied to the central core of the system will produce a badly needed example of a structure that is significantly easier to understand.

The Security Kernel

The basic structural concept for organizing a certifiably secure system is that of a security kernel. A security kernel is a minimal, protected central core of software whose correct operation is necessary and sufficient to guarantee enforcement within a system of the security model. 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 certify the security properties of the system.

The minimum size and complexity of the security kernel required to support a particular system is limited by the security model that the system must match. The complexity of the patterns of access constraints to be enforced is an obvious factor. The complexity of the abstract objects and operations in terms of which the model is expressed is also important, for a correct implementation of all such objects and operations becomes

a prerequisite to security, and thus such implementation becomes part of the security kernel. If the desired security model can be expressed only in terms of the topmost layers of objects and operations in a system, then the security kernel would encompass almost all the software in the system. A primary goal when trying to produce a certifiably secure system is the isolation of the smallest, simplest (3) security kernel that is capable of supporting the full functionality of the system.

A characterization of the kind of mechanisms that will be included in a security kernel can be gotten by viewing a security model 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), for a common mechanism is required if one computation is to influence another. A mechanism is common to two computations if it contains some set of data items whose value one computation can influence and the other can notice. The influence and notice may be very direct--one writes into a data item and the other reads it--or quite indirect--the invocation of a procedure by one somehow

(3) Unfortunately, no objective measure of overall simplicity is known.

alters the procedure's internal state so that the outcome of a later 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. It is precisely the existence of common mechanisms that allow one user the possibility of exerting unauthorized influence over the computations or data of another. Malicious users must exploit flaws in common mechanisms to work their will. To prevent such malicious activity it is the common mechanisms that must be certified to contain no exploitable flaws, and once certified must be protected 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 required in the system.

Given a security model, designing a security kernel requires deciding what mechanisms should be in the kernel and then organizing the structure of the kernel to make matching it with the model as easy as possible. It is these two engineering problems that we are addressing with respect to Multics.

At first glance, deciding which mechanism to include in the security kernel seems straightforward. A mechanism should be in the kernel if implementing that mechanism as an unprotected part

of a user's computation would allow the user to construct a program that violated the security model. The problem with this simple test, however, is defining the mechanisms to be tested. It is very easy to combine in what appears to be a single mechanism functions that need to be protected and those that do not. The test places all of such a mechanism in the kernel. The challenge is to define mechanisms that do not combine functions needing protection with those that do not. There appears to be no systematic approach to generating such mechanisms. The first major task of engineering a security kernel for Multics is carefully rethinking each mechanism in the current Multics supervisor to untangle the functions that must be implemented in the security kernel from those that can be implemented as an unprotected part of each user's computation. (4)

The problem of structuring a kernel to facilitate establishing a correspondence with a security model is not understood very well. The difficulties start at the level of choosing design principles with which to generate the kernel's internal structure. It seems apparent that enforced protection barriers can be used to good advantage within the kernel to modularize the task of matching the kernel to the model. But there are many ways to generate this modularity. To illustrate

---

(4) Note that a usual reason for including a mechanism in a supervisor, to obtain protection from accidental breakage of the mechanism by errors in user code, is not in itself sufficient reason to include a mechanism in a security kernel. Non-kernel mechanisms may be made breakproof by protecting them in other domains that are private to each user's computation.

two possibly conflicting approaches to structuring a kernel, imagine that the security model is expressed as a set of security properties, each of which must be met. One technique of modularization 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 model, an independent verification of each property is required, but each involves only a subset of the domains in the kernel. Another technique is to ignore any structure suggested by the security properties and divide the kernel into domains according to a principle like Parnas' notion of information hiding [11], so that each domain has a simple interface behavior specification. To verify each of the security properties may involve all the kernel domains, but once each domain is 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. The second major task of engineering a security kernel for Multics is finding a satisfactory way to structure the kernel by experimenting with different approaches and subjectively evaluating their impact on kernel complexity. Some of the ideas being explored are discussed later.

While a security kernel contains all the mechanisms that must be considered to certify a system, a correct kernel does not guarantee the integrity of all computations or stored data in a

system. Non-kernel software still can cause undesired release of information, modification of information, or denial of use. But if the kernel is correct, then these undesired results will not be unauthorized. To understand the meaning of this distinction, consider the non-kernel software as grouped in four catagories.

First, there are the system-provided programs that execute as part of user computations. These include the library subroutines, compilers, and applications packages available in most systems plus 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 non-writeable code that embodies their algorithms, since 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. They may contain errors, but these errors can by triggered only by the actions of the computation that they might damage as a result. By presuming that the system programmers who constructed them are not malicious and did not willfully plant "trojan horses," it seems justified to assume that the mistakes caused by these system-provided programs will decrease in time as all normally used functions are exercised, and that the security 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, these are not susceptible to willful exploitation by other users. In any case, a user unsatisfied with their trustworthiness may choose not to use them, substituting his own programs.

The second catagory 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 would be providing tools to aid the user in 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 security of the borrower's data. Because they will execute with all the access authority of the borrower's own programs, they can contain "trojan horse" code maliciously constructed to cause a results undesired by the borrower. (5) A user should only borrow programs from another when the borrower has reason to trust the lender. The inclusion of security kernel facilities to support user-constructed protected subsystems provides a tool to reduce the potential damage such a borrowed trojan horse can do, but a user initiated certification of the borrowed program is the only complete protection against this threat.

The fourth category is common mechanisms set up among a

---

(5) Note that 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 lent program itself. Even if the program is non-writable when lent, it was written by the lender when constructed.

subgroup of system users by their mutual consent 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 uncertified 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.

In considering these four categories, it is apparent that the essential mechanism to certify is the common mechanism of the security kernel, for every user of the system is forced to rely upon it. A certified 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 security kernel prevents unauthorized activities as defined by the security model for a system. Not all undesired results, however, are the result of unauthorized activities. (6)

---

(6) A fifth catagory of non-kernel 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 be certified, as well as the kernel, a troubling thought since the compiler may well be larger than the kernel. In the case of the compiler for the kernel, however, certification may be less of a problem than for the kernel. The kernel needs to work correctly for all

The Multics Kernel

This section describes some concrete examples of the specific strategies being employed to carry out the general research plan outlined in the previous sections. The intention is to communicate more precisely the spirit of the work rather than to provide a thorough discussion of the structure or insights resulting from the various activities. The specific results of the individual activities are being communicated when appropriate in other reports. The activities underway or planned can be broken into four interrelated categories: review, removal, simplification, and partitioning. Each will be discussed in turn.

The review category covers efforts to understand better the specific problems of the current Multics supervisor. In addition to trying to understand the reasons for the size and complexity of the current supervisor, an effort is being made to identify and correct existing security flaws. A list of all known Multics security flaws is maintained. Each flaw reported is analyzed to determine how it happened, how it can be fixed, and how similar flaws can be avoided in the security kernel being developed. So far, all of the flaws uncovered by the review activities are isolated and easily repaired. No major design flaws have been

possible inputs; the compiler need compile correctly only the specific programs of the kernel--not all possible programs. Thus, the compiler's effect on the kernel can be certified by comparing the source code "model" for each kernel module with the compiler-produced object code "implementation", a task much simpler than certifying the compiler correct for all possible source programs.

found.

The second category of activity is removing from the supervisor and placing in the user domains of a process those mechanisms not implementing functions of information sharing, interprocess communication, or physical resource management, i.e., those functions not required to be implemented as common mechanisms. In many cases removal involves undoing a pattern caused by a performance characteristic of the Multics implementation for the Honeywell 645 computers. For that older machine the multiple protection domains of a process, the so called protection rings, were simulated in software and cross-ring calls were quite expensive. Thus, a call that went from a user ring in a process to the supervisor ring cost much more than a call which did not change protection environments. The result was an effective pressure to include many functions in the supervisor that did not need to be implemented as part of a common mechanism. The reason for this pressure can be seen by considering two procedure modules, A and B, in the supervisor. Imagine that a single invocation of A (by a user procedure) can result in a flurry of calls from A to B. If calls that change the ring of execution of a process are more expensive than calls that do not, then there is a clear performance cost in placing the supervisor boundary between A and B, even if only B need be part of the protected, common supervisor.

The new hardware base for Multics, the Honeywell 6180, implements the protection rings in hardware. One result is that

calls from one ring to another now cost no more than calls inside a ring. Thus, the performance penalty associated with supervisor calls has been removed, and many modules included in the supervisor for performance reasons rather than protection reasons now can be removed, (7)

The actual removal activities are much more complex than suggested by the example of the previous paragraph. In most cases the common and private parts of a facility are not so neatly packaged in separate procedures, but are intricately intertwined in the same supervisor procedures and data bases. Insight and ingenuity are required to separate the private and common parts of a mechanism, leaving a reasonable interface. The key problem is finding the proper decomposition of the supervisor into protected and not protected primitive functions.

Most initial removal activities have been centered on the file system. In a project now completed the functions of dynamic intersegment linking and directing the search of the file system to satisfy a symbolic reference have been removed from the supervisor [12,13]. This project is notable for two reasons. First, it removed an especially 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;

---

(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.

the chances of such a complex "argument", if maliciously malstructured, causing the linker to malfunction while executing in the supervisor were demonstrated to be very high by numerous accidents. The complexity is apparent in that the linker's removal eliminated 10% of the gate entry points into the supervisor. The second interesting result of the linker's removal was the demonstration that linking procedures together across protection boundaries, i.e., rings, could be done without resort to a mechanism common to both protection regions.

A second completed project relating to the file system is the removal from the supervisor of the facilities for managing the association between reference names and the segments in the address space of a process [14]. Removal of this naming mechanism from the supervisor required that a data base central to the management of the address space, the known segment table, be split into a private and a common part, and that the supervisor learn to lie convincingly on occasion about the existence of certain file system directories. The result of the removal is a reduction by a factor of ten in the size of the protected code needed to manage the address space of a process. Another result is 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 is used. The algorithms for following a tree name through the file system hierarchy to locate the named element are thus removed from the supervisor to be

implemented by procedures executing in the user ring. (The actual file system hierarchy remains protected inside the supervisor.) The linker and reference name removal projects together reduce the number of user-available supervisor entries by approximately one third.

A removal project under investigation is changing most of system initialization from executing inside the supervisor each time the system is started to executing once in a user environment of a previous system. The idea is to produce on a system tape a bit pattern which, when loaded into memory, manifests a fully initialized system, rather than letting the system bootstrap itself in a complex way each time it is loaded from a tape containing the separate pieces. One pattern of operation may be much simpler to certify than the other.

A final example of a removal project is the exploration of a recently-realized equivalence between the mechanics of entering a protected subsystem and the mechanics of creating a new process in response to a user's log in. The goal is to make a single mechanism do both tasks, with the result that the large collection of privileged, protected code used to authenticate and log in users would become non-privileged code.

The third category of activity is simplifying those mechanisms that must remain in the kernel. Such activities can reduce both the size and the complexity of the kernel. Simplification activities cover a broad range. In some cases a piece of the kernel can simply be eliminated because its function

can be duplicated by another kernel mechanism. For example, the possibility of replacing all mechanisms for performing external I/O (to terminals, tape drives, card readers, card punches, and printers) with the ARPA Network attachment is being explored. This would remove from the kernel a large bulk of special mechanisms for managing the various I/O devices, leaving behind a single mechanism for managing the network attachment. Using network technology to provide the only path for external I/O to Multics appears feasible. Internal I/O functions (for managing the virtual memory, performing backup, and loading the system) would still be managed in the kernel.

Another example of simplification involves a less obvious duplication of mechanisms. A new buffering strategy for input from the network has been devised which, by utilizing the virtual memory, provides a core resident buffer which appears to be of infinite length. The infinite buffer scheme is much simpler than the old circular buffer which had to be used over and over again, with attendant problems of old messages not being removed before a complete circuit of the buffer was made. The old buffer scheme was really providing a special purpose storage management facility, and the simplification was to use the standard storage management facility of the system--the virtual memory--for this function.

Several specific simplification projects involve the implementation and use of processes. One project, now nearing the end of the design phase, is a reimplementation of processes

using two layers of mechanism. (8) This new design simplifies the interaction of the process implementation with the virtual memory management machanisms, and simplifies the base-level interprocess communication mechanisms of the system. The first level multiplexes the processors into a larger fixed number of virtual processors. Because the number of virtual processes is fixed, this first layer need not depend on the facilities 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, while 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. The proposed new base-level interprocess communication facility has the property that its use can be controlled with the standard memory protection mechanisms of the kernel.

The implementation of certain kernel mechanisms as asynchronous parallel processes, as implied above, represents a simplification of the present system design which forces many supervisor mechanisms into sequential algorithms. The virtual memory mechanisms for moving pages among the three levels of the memory hierarchy is a good example. Whenever a missing page fault occurs in a process, the fault handler attempts to initiate

---

(8) This idea is being explored by others as well [4,15].

the transfer of the desired page from bulk store or disk to primary memory. This can only be done if a free primary memory block is available. If not then the fault handler first 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 by the fault handler. With the current system design, this complex series of steps occurs sequentially with page control executing in the process which 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 runs in a loop making 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 is awakened to transfer pages to bulk store. Another keeps space free on the bulk store by moving pages to disk when required. The primary memory freeing process is activated by wakeups from processes that have taken a page fault and discovered a lack of free primary memory blocks. The bulk store freeing process is driven in a similar manner by the primary memory freeing process. 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. The overall structure looks as though it will be much simpler than that currently employed.

Another application of parallelism in the kernel being considered is handling interrupts. Each interrupt handler will 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 will simply turn each interrupt into a wakeup of the corresponding process. By virtue of being full-fledged processes, the interrupt handlers can use the normal system interprocess communication mechanisms to coordinate their activities with one another and the user process, greatly simplifying their structure.

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

The final category of activity is partitioning the kernel into differently protected pieces to modularize the job of certification. While the specific projects in this category are less well developed than those for other categories, two techniques for partitioning seem worth exploring. The first is dividing the part of the kernel that is part of each process into multiple layers in different rings of protection. For example, the bottom layer might 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 absolute compartmentalization of users and

stored information be implemented at the bottom layer (according to the Mitre model mentioned earlier), and mechanisms to allow controlled sharing within the compartments be implemented at 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 each compartment.

The second partitioning technique under investigation is using the protection rings in the kernel processes that implement resource management algorithms to separate the policy component from the mechanism component. (9) For example, the process described earlier that removed pages from primary memory could be arranged 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

---

(9) Separation of policy from mechanism is a structural principal that has been explored by many others [16,17,18].

overwrite another, for 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 certified as the rest of the kernel. It appears that the idea of separating policy from mechanisms applies to all resource management algorithms.

The above sample of activities is characteristic of the initial thrust of this research effort.

Conclusion

This paper has presented the plan for a research project aimed at reducing the unnecessary complexity exhibited by available general-purpose systems, and has described a sample of the specific strategies being employed. Reduced complexity is a key component of developing systems whose security properties can be certified, no matter what means are used to match the system implementation to a model of its security properties. In addition, understanding simplier structures with which to implement the features desired in general-purpose systems will contribute to the development of the methodical system construction techniques that lead to the ultimate solution of the computer security problem.

Acknowledgements

To be supplied.

References

[1] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," IEEE Proc., to appear, Sept. 1975.

[2] R. R. Linde, "Operating System Penetration," AFIPS Conf. Proc. 44, pp. 361-368, NCC 1975.

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

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

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

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

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

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

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

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

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

[12] P. A. Janson, "Removing the Dynamic Linker from the Security Kernel of a Computer Utility," M.I.T. Project MAC Technical Report MAC-TR-132, June 1974.

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

[14] R. G. Bratt, "Minimal Protected Naming Facilities for a Computer Utility," M.I.T. Project MAC Technical Report, to appear, 1975.

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

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

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

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