The security kernel approach provides controls that are effective against most internal attacks—including some that many designers never consider.

Security Kernel Design and Implementation: An Introduction

Stanley R. Ames, Jr., and Morrie Gasser, The Mitre Corporation

Roger R. Schell, DoD Computer Security Center

Providing highly reliable protection for computerized information has traditionally been a game of wits. No sooner are security controls introduced into systems than are penetrators finding ways to circumvent them. Security kernel technology provides a conceptual base on which to build secure computer systems, thereby replacing this game of wits with a methodical design process. The kernel approach is equally applicable to all types of systems, from general-purpose, multiuser operating systems to special-purpose systems such as communication processors—wherever the protection of shared information is a concern.

Most computer installations rely solely on a physical security perimeter, protecting the computer and its users by guards, dogs, and fences. Communications between the computer and remote devices may be encrypted to geographically extend the security perimeter, but if only physical security is used, all users can potentially access all information in the computer system. Consequently, all users must be trusted to the same degree. When the system contains sensitive information that only certain users should access, we must introduce some additional protection mechanisms. One solution is to give each class of users a separate machine. This solution is becoming increasingly less costly because of declining hardware prices, but it does not address the controlled sharing of information among users. Sharing information within a single computer requires internal controls to isolate sensitive information.

Continual efforts are being made to develop reliable internal security controls solely through tenacity and hard work. Unfortunately, these attempts have been uniformly unsuccessful for a number of reasons. The first is that the operating system and utility software are typically large and complex. The second is that no one has precisely defined the security provided by the internal controls. Finally, little has been done to ensure the correctness of the security controls that have been implemented.

The security kernel approach described here directly addresses the size and complexity problem by limiting the protection mechanism to a small portion of the system. The second and third problems are addressed by clearly defining a security policy and then following a rigorous methodology that includes developing a mathematical model, constructing a precise specification of behavior, and coding in a high-level language.

The security kernel approach is based on the concept of the *reference monitor*, an abstract notion adapted from the models of Butler Lampson.[1] The reference monitor provides an underlying security theory for conceptualizing the idea of protection. In a reference monitor, all active entities such as people or computer processes make reference to passive entities such as documents or segments of memory using a set of current access authorizations (Figure 1). Of particular importance is that *every* reference to information (e.g., by a processor to primary memory) or change of authorization must go through the reference monitor.

The security kernel is defined as the hardware and software that realize the reference monitor abstraction. To successfully implement a security kernel, we must adhere to three engineering principles: (1) *completeness*, in that all access to information must be mediated by the kernel; (2) *isolation*, in that the kernel must be protected from tampering; and (3) *verifiability*, in that some correspondence must be shown between the security policy and the actual implementation of the kernel. The completeness and isolation requirements are best addressed with an adequate hardware foundation. A formal development methodology can be a powerful tool for addressing the verifiability requirement.

Schell first introduced the security kernel concept in 1972 as "a compact security 'kernel' of the operating system and supporting hardware such that an antagonist could provide the remainder of the system without compromising the protection provided." In 1974, Mitre tested the hypothesis that such a security kernel could actually be constructed. The first security kernel consisted of less than 20 primitive subroutines that directly managed the physical resources and enforced protection constraints. The entire security kernel contained fewer than a thousand compilable high-level language statements and ran on a DEC PDP-11/45.

To demonstrate this kernel, Mitre also constructed a simple, experimental operating system along with applications for a practical military example. The operating system had a hierarchical file system, cooperating processes with controlled information sharing, and interfaces to a few interactive terminals. The operating system and applications were outside the kernel and could not impact the information protection provided by the kernel.

Since this initial prototype effort, a number of research efforts have dealt with the issues of security kernel construction and verification. Today, a few security-kernel-based products are being introduced commercially. One such system, the Honeywell Secure Communications Processor (Scomp), is discussed by L. Fraim[2] (see the article in this issue); others are surveyed by C. Landwehr[3] (also in this issue).

Basic principles

The first step in developing a kernel-based system is to identify the specific set of protection policies to be supported. A given system is "secure" only with respect to some specific policy. In a computer system, a well-formed protection policy should identify all the permissible modes of access between the active entities, or subjects, and the passive entities, or objects. The external policy that corresponds to the people, paper, and methods of accessing information in the real world must be interpreted in a way that allows the policy to apply to the internal entities of the computer system.

This requirement—that policy be precisely defined—is a primary distinction between a security-kernel-based system and several other efforts to develop security-relevant operating systems, such as "capability" machines.[4] These other systems tend to strive for general-purpose protection, yet do not have any definitive criteria for what is security relevant. The mechanisms in these systems essentially provide a computer with special security features. By contrast, the security-kernel approach explicitly addresses both policy and mechanism. If general-purpose protection mechanisms are well-defined and augmented to enforce a specific policy, however, they can provide an underlying base for subsequent security kernel construction.

A formally defined security model. We need to define two types of policy: nondiscretionary and discretionary. A *nondiscretionary* policy contains mandatory security rules that are imposed on all users. A *discretionary* policy, on the other hand, contains security rules that can be specified at the option of each user.

The protection policy enforced by a security kernel is encapsulated in a set of mathematical rules that constitute a formal security model. Both discretionary and nondiscretionary policies must be addressed by the rules of the model.

In determining whether the model of a policy is sufficient, we need to consider two key issues, which occasionally have been a source of misunderstanding. First, a model of a policy must define the information protection behavior of the system as a whole. Merely modeling distinct operations with respect to individual assertions about a protection mechanism does not indicate much about overall system security and can, in fact, be misleading. Second, a model of a policy must include a "security theorem" to ensure that the behavior defined by the model always complies with the security requirements of the applicable policy.

The model enforced by most security kernels has been derived from early security kernel work at Mitre[5] and Case Western Reserve University.[6] Commonly referred to as the Bell and LaPadula model, this model provides rules for preventing unauthorized observation and modification of information. By representing the security kernel as a finite state machine, these rules define allowable transitions from one "secure" state to the next.

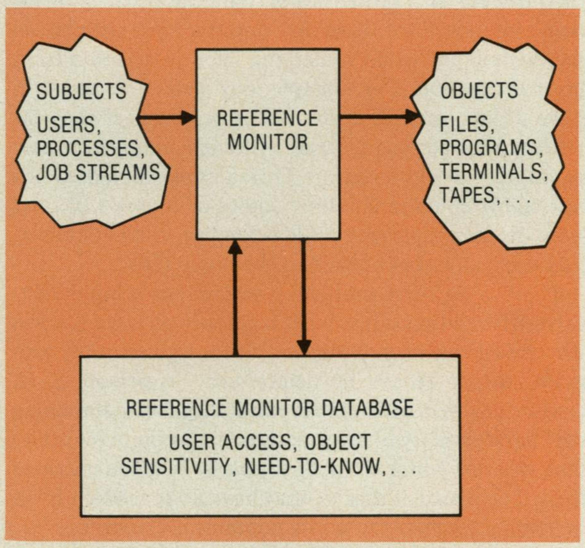

SUBJECTS
USERS, PROCESSES, JOB STREAMS
REFERENCE MONITOR
OBJECTS
FILES, PROGRAMS, TERMINALS, TAPES, . . .
REFERENCE MONITOR DATABASE
USER ACCESS, OBJECT SENSITIVITY, NEED-TO-KNOW, . . .

Figure 1. Reference monitor. The security kernel approach is based on the concept of a reference monitor, in which active entities (subjects) make reference to passive entities (objects) using a set of access authorizations (reference monitor database).

Within the model, each subject and object of the reference monitor is given a security identifier termed an *access class*. The access classes of subjects and objects are compared at each state transition to determine whether a subject is allowed to access an object. By organizing the access classes in the form of a mathematical structure called a lattice, a wide range of potential policies can be supported. The lattice defines the relations among access classes, allowing us to determine whether one access class is less than, greater than, equal to, or disjoint from (not comparable to) another. Examples of environments in which access classes form a lattice include the privacy protection compartments of medical data, financial data, criminal records, and the hierarchical government security classifications.

The protection policy enforced by a security kernel is encapsulated in a set of mathematical rules that constitute a formal security model.

Of the model's nondiscretionary rules, two are fundamental. The first, called the *simple security condition*, states that a subject cannot observe the contents of an object unless the access class of the subject is greater than or equal to the access class of the object. This simple security condition prohibits users from directly viewing data that they are not entitled to see.

The second basic nondiscretionary rule, the **-property* (pronounced "star" property) rule, helps to prevent all illicit indirect viewing of objects. It stipulates that a subject may not modify an object unless that object's access class is greater than or equal to the access class of the subject.

The purpose of the *-property is to explicitly address the problem of "Trojan horse" software, which as the name implies is software that appears legitimate but in fact is designed to do something illicit in addition to its normal function. For example, any generally used software utility, such as a text editor or compiler, has the potential for accessing a user's files in a manner the user might not have intended. A Trojan horse implanted in a text editor could make illicit copies of a user's file and store the information in a file belonging to an unauthorized user, all unbeknownst to the original user.

The Trojan horse problem is serious when highly sensitive information is involved, especially on large systems where the programmer responsible for a given utility program cannot always be determined. A person is, of course, charged with the responsibility for maintaining the confidentiality of information, but a computer utility such as a text editor cannot necessarily be given the same trust. The reason is that we may have no practical way to determine whether the utility contains a Trojan horse.

With the *-property, information cannot be compromised through the use of a Trojan horse. Under this rule, the program operating on behalf of one user cannot be used to pass information to any user having a lower or disjoint access class.

The simple security condition and the *-property are primarily to prevent the unauthorized disclosure of information, but the model also includes integrity properties to protect information from improper alteration. Integrity rules prevent subjects with a given integrity class from modifying objects of higher integrity or being affected by objects of lower integrity.

The nondiscretionary rules of the model do not provide a protection policy that distinguishes different users within the same access class. Discretionary rules are included in the model to provide that type of protection policy. The discretionary rules of the Bell and LaPadula model allow authorized users and programs to arbitrarily grant and revoke access to information based on user names or other information. Since discretionary controls are more or less arbitrary, we cannot make very many absolute statements about the movement of the information. In particular, the Trojan horse attack is more difficult to address under these controls than under nondiscretionary controls. Therefore, the latter, being much stricter, always takes precedence.

In addition to the threats of improper disclosure or modification of information, we have a threat known as *denial of service* (e.g., crashing the system or making it unresponsive), which most security models such as the Bell and LaPadula model do not explicitly address. While a kernel-based system is likely to withstand this threat at least as well as any conventional system, rules dealing with denial of service are more difficult to formalize in a model.

Faithful implementation. The mathematical model aids in identifying the types of functions that a kernel should provide. It does not, however, specify the design for the applications interface to the kernel. To bridge the gap between model and implementation, the development process must be broken into small steps. One common technique is to apply a hierarchy of abstract specifications to the design of the security kernel. For each step, it is important to demonstrate security so that we have confidence in the security of the final system.

We can use numerous formal and informal methods to demonstrate security with varying degrees of confidence—the reference monitor concept does not mandate any one approach. However, early in the formulation of the kernel approach, we recognized that formal specification and mathematical verification had the potential for providing real proof that the implementation of the kernel faithfully followed the rules of the model. These formal methods have since been applied to various degrees in demonstrating the correspondence between the model, the hierarchy of specifications, and the high-level language implementation (Figure 2).

As with any operating system design effort, preparing the specifications is a creative activity, molded by the particular design and security goals of the system. The most abstract kernel specification defines all the kernel's interface characteristics. We can use this high-level specification to judge functionality as well as to demonstrate that the interface preserves the rules of the model. Once the interface functionality is specified abstractly and its security properties are precisely established, we can expand the functionality by gradually introducing more implementation detail. This process is done without affecting the validity of the security properties already established. (For examples of systems that use formal specification techniques, see the article by Landwehr in this issue.)

Three classes of formal verification techniques have been applied to different stages of kernel development (Figure 2), and several techniques are available within each class. The first class is used to prove that the kernel's intended behavior, as described in the formal high-level interface specification, is secure with respect to the policy model. One common technique, *security flow analysis,* is a relatively simple way to identify and analyze information flows in a specification.[7] Note that only the security of the interface specification must be demonstrated, not the more difficult problem of its functional "correctness," since functional properties, most of which are not security related, are not addressed by the model.

In the second class of formal verification techniques, we verify the correspondence or correctness of mappings between any intermediate specifications in the hierarchy and the interface specifications. Finally, a third class of verification techniques, the most traditional way to prove correctness, shows that the kernel implementation corresponds to its specification.

Cheheyl et al. have documented a survey of current verification systems covering most of these techniques, along with their application to Department of Defense security policy.[8] Walker et al. describe an example of a formal specification and verification.[9]

Implementation considerations

To successfully realize a kernel-based system, we must take into account architectural and engineering considerations that may not be encountered in the development of other systems. Although the kernel approach can be applied to all types of systems, these considerations are best illustrated in the context of a general-purpose operating system with online, interactive users (Figure 3). The kernel, as already noted, provides a relatively small and simple subset of the operating system functions. The kernel primitives are the interface of this subset to the rest of the operating system (generally referred to as the supervisor). In turn, the supervisor primitives provide the general-purpose operating system functions used by the applications.

Kernel/supervisor trade-offs. An operating system is usually broken down into functional areas, such as process management, file system management for segments, and I/O control. Within each area, some functions are clearly security relevant and must be in the kernel, while some are not. The rules of the policy model help to clearly identify which functions are security relevant.

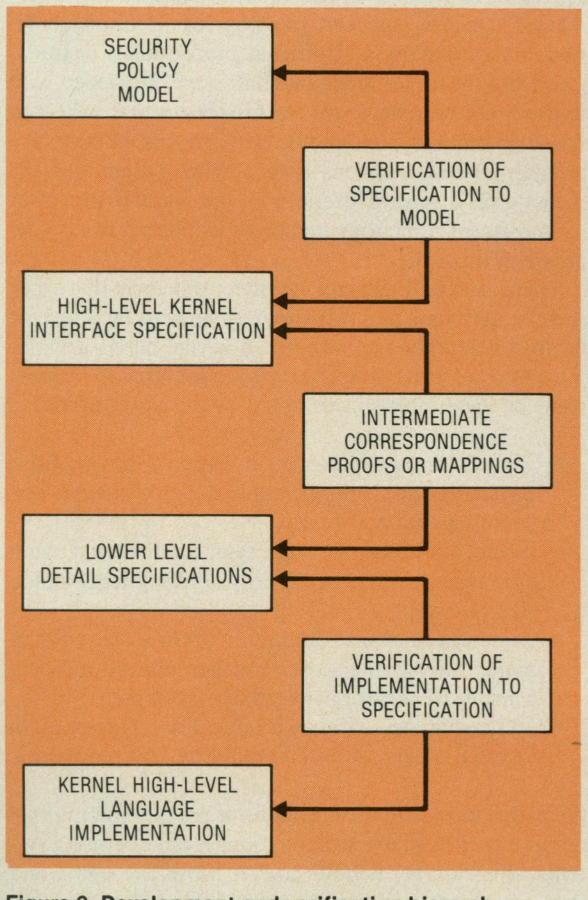

SECURITY POLICY MODEL
VERIFICATION OF SPECIFICATION TO MODEL
HIGH-LEVEL KERNEL INTERFACE SPECIFICATION
INTERMEDIATE CORRESPONDENCE PROOFS OR MAPPINGS
LOWER LEVEL DETAIL SPECIFICATIONS
VERIFICATION OF IMPLEMENTATION TO SPECIFICATION
KERNEL HIGH-LEVEL LANGUAGE IMPLEMENTATION

Figure 2. Development and verification hierarchy.

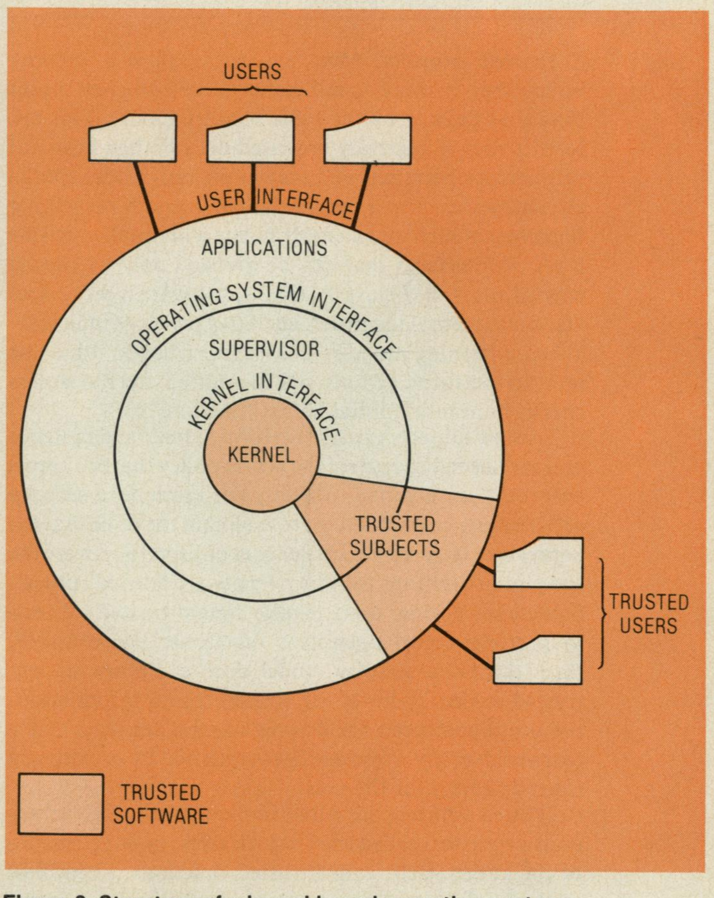


Figure 3. Structure of a kernel-based operation system.

The kernel must handle the parts of an operating system that manage resources, such as memory and disk space, shared by multiple users. These parts are in the kernel because the model requires that these resources be virtual to hide their location from untrusted (nonkernel) software. The functions that provide useful common utilities but do not manage anything shared among users and those that address denial of service are outside the scope of the security policy and can generally be in the supervisor.

In practice, we can usually apply the details of the model and a particular security policy to determine what must be in the kernel and what must be in the supervisor. However, issues such as performance and functionality often force us to consider engineering problems that necessitate putting non-security-related functions in the kernel. We might have trouble, for example, separating the operating system's file-name interpretation mechanism, which may not be security relevant, from the kernel's file management system. We must constantly make trade-offs among performance, functionality, and complexity when designing a kernel-based system. Trade-offs are especially important when the system is emulating a previously existing operating system whose functions may not be easy to reallocate.

The kernel must handle the parts of an operating system that manage resources shared by multiple users.

Trusted subjects. Most systems require a security policy that is more specifically tailored to their needs than that defined by the basic security model. This tailored policy is generally exercised on a limited basis for infrequent operations and may apply only under special circumstances or to a special class of users. A kernel that implements such an extended policy will usually provide a set of interfaces that can be invoked only by certain *trusted subjects* (Figure 3), that is, software it recognizes via some internal identifier, such as a privilege indicator. When a running program has such privileges, it may be able to perform actions not permitted by the access checks built into normal kernel functions.

Trusted subjects usually perform system maintenance and are needed to control the access policy that the kernel enforces for untrusted subjects. For example, a security officer must be given access to maintain the table that the kernel uses to specify access classes of users. Software the security officer uses for this purpose is a trusted subject. Sometimes normal users invoke certain trusted subjects to perform security-sensitive functions. For example, since the basic security model does not allow an untrusted subject to lower the access class of information, the occasional need for downgrading a segment that a user accidentally overclassifies is satisfied by providing a trusted subject for the user.

Trusted subjects are often implemented as asynchronous processes, called trusted processes, or as extensions of the kernel itself, called trusted functions. Regardless of the implementation technique, trusted subjects must adhere to the same engineering principles as the kernel if the security policy is to be correctly implemented. Other than the implementation technique, the only difference is the specific security policy enforced.

Hardware/software features. The kernel approach may need considerable hardware support to achieve adequate performance. The amount of hardware support is bounded by two extremes. At one extreme, the kernel can be built entirely in software on any conventional machine so that the kernel runs as a pure interpreter, executing and checking every user instruction and permitting no direct user execution of hardware instructions. In this case, no particular security demands are placed on the hardware architecture other than the requirement to execute the kernel software correctly. At the other extreme, all the kernel functions can be implemented as hardware instructions; in this case the hardware architecture would be completely responsible for security. As with the supervisor/kernel trade-offs, the specific choices are heavily influenced by the trade-offs among complexity, size, and performance. In this article, we examine only a pragmatic middle ground, one close to a traditional view of the functional division between the hardware and the operating system software.

The hardware features and software mechanisms necessary for a kernel-based operating system to perform adequately are sophisticated but not exotic. The specific hardware features desirable for a kernel-based system are provided in many (but by no means all) modern computer architectures, from microprocessors to mainframes. Several past and ongoing kernel implementations have resulted in significant performance degradation from the lack of adequate hardware. However, with appropriate hardware, we see no reason that a kernel-based operating system should perform any worse than a non-kernel-based system with similar capabilities. R. Schell gives a specific example of the features necessary to implement a microprocessor-based kernel in another article in this issue.[10]

There are four general architectural areas in which specific hardware and software mechanisms have proved useful or necessary to support a kernel-based general-purpose operating system (for special-purpose kernels, some of these mechanisms might be less appropriate):

- explicit processes—efficient support for multiple processes (multiprogramming) and interprocess communication;
- memory protection—large segmented virtual memory, access control to memory, and explicitly identified objects;
- execution domains—minimum of three states or domains (user, supervisor, and kernel) and efficient transfer of control between domains; and
- I/O mediation—control of access to I/O devices, to external media, and to memory by I/O processors.

Although most of these mechanisms are familiar internal components of many current commercial operating systems, their impact is quite hidden from the users, and they are usually of interest only to system designers.

Explicit processes. The reference monitor's notion of a subject is traditionally realized in most operating systems as a process that operates as a surrogate for some user. The user's identification and access class must therefore be represented within the system as nonforgeable (and unalterable) identifiers tied to each process. The identifiers become the basis for making access decisions with respect to discretionary and nondiscretionary policies.

By *process* we mean the activity of a processor carrying out the computation specified by a program, the processor being either a computation or I/O processor. For information protection to be meaningful, an on-line environment must of course include multiple users, so the kernel must support multiple simultaneous processes. This requirement mandates that the kernel save and restore the representation of a process in execution, otherwise known as the state of the processor. Depending on when a process is allowed to be suspended, this state may include the internal state of the CPU, the user-visible processor registers, or merely the instruction counter. In addition, the architecture must provide a means of saving and restoring a definition of the accessible information (i.e., the address space) distinct for each process. The address space is typically defined by a set of descriptors, as we discuss later in the section on memory protection.

Because of the multiplicity of simultaneous processes, a kernel-based operating system typically has a large number of process switches, and an efficient process-switching mechanism is desirable. This mechanism can be supported in a number of ways: we can use high-speed memory instead of explicit processor registers, or we can load and store processor registers as a block or even have several independent sets of registers in the processor. The efficient switching of address spaces can be aided by using a descriptor base or root register instead of copying memory descriptor tables, or by simultaneously retaining several sets of descriptors.

In addition to the multiprogramming support, we need direct support for interprocess communication. Particularly for multiprocessor configurations, we need a race-free communication mechanism (e.g., read-alter-rewrite memory access) as well as a processor-to-process-or interrupt capability. Note that an I/O initiation instruction is primarily a mechanism for directing an interrupt to an I/O processor. More sophisticated hardware mechanisms, such as operations to assist process synchronization, can also contribute to kernel simplicity and performance.

Memory protection. The reference monitor abstraction of a storage object is usually realized by memory, and this realization is constrained by the principle of completeness identified earlier. Clearly, some fundamentally interpretive mechanism is needed to completely mediate all access to memory. Virtual memory is commonly used to accomplish the needed mediation. In a virtual memory system, some form of descriptor is used to control the access to memory. There must be no way outside the kernel for a process to access memory without using a descriptor.

With a reference monitor, all information within the system must be represented in distinct, identifiable objects. In all but the simplest case, the virtual address space of a process includes more than one object, each with distinct logical attributes such as size, access mode, and access class. This logically distinct memory is commonly called a *segment.*

With a reference monitor, all information within the system must be represented in distinct, identifiable objects.

Hardware-supported segmentation of virtual memory is the underlying mechanism to support this concept. Each segment is identified by a descriptor that controls the virtual-address-mapping hardware. The descriptor contains some logical attributes as well as a physical base address and a segment size (or bound) to distinguish each segment. Complete access mediation is provided, since for each access to virtual memory the hardware must interpret the relevant descriptor.

Two factors are quite important in supporting an efficient and simple secure system: (1) each process should be able to have a relatively large number of independent segments, and (2) any segment should have a wide range of possible sizes. These requirements tend to result in a large number of segment descriptors for each process. A high-speed associative cache memory can be useful to speed address translations without requiring software to reload descriptors on each process switch.

Architectures in which a process's view of virtual memory is not segmented but is simply a large, linear address space, with no portion shared by other processes, are often compatible upgrades to older systems without virtual memory. Although kernel-based systems can be and have been built on such architectures, considerable flexibility (and resulting performance) can be lost because of the inability to directly address different segments with differing access rights.

To implement typical discretionary and nondiscretionary access policies of the reference monitor, the segment descriptor must support distinct access modes of at least null, read, and read-write for each segment.

Overall system performance can often be enhanced by including a referenced and modified flag for each block of physical memory. This enhancement permits more efficient operating system memory management schemes when information must be moved back and forth between primary memory and secondary storage.

The segment descriptors are, of course, managed by the security kernel software, although much of the actual mediation of the reference monitor is performed by the address-mapping hardware. Since address mapping requires an examination of the descriptors, the hardware can conveniently check access using information in the descriptor at the same time with no additional performance penalty. The security kernel software enforces reference monitor authorizations by controlling the access mode specified in the descriptors for all segments of each process.

Even if all access to segments is fully controlled, the kernel designer may encounter a major pitfall: the possibility that information will be leaked unintentionally through the use of *control information.* Control information consists of items that are not memory objects in the usual sense but are shared repositories of information. They include items maintained within the kernel database, for example, file names and attributes, system variables such as the number of users logged in, and the sizes of message queues. Although these items are not within the hardware-supported virtual address space, they are objects to the reference monitor, and thus the kernel must mediate access to each of them in the same way that it controls access to segments. Access to most of these items is usually done interpretively through explicit kernel calls, rather than through hardware.

Any accidental leakage of information through the use of control information should, of course, be detected by appropriate design and verification techniques. The pitfall is the possibility that some fundamental aspect of the design is based on (and in fact may depend on) this leakage. This issue must be recognized early in the design because at a late stage of system development, the removal of undesired leakage channels can be one of the most difficult tasks a kernel designer can encounter.

Execution domains. Execution domains are essential to the isolation and protection of the security kernel mechanism. The total address space of a process includes the programs and data of the security kernel, since these must clearly be accessible when the security kernel functions are invoked. Yet, the kernel also requires a distinct execution domain so that a process can access some objects (most notably the segment descriptors) only when executing in the kernel itself.

The simplest and most common domain structure is made up of two hierarchical domains implemented by privileged and nonprivileged modes of processor execution. The privileged domain contains only the kernel. Although two domains are sufficient to protect the kernel, the supervisor would have to reside in the same domain as the applications software—a serious limitation. Operating systems traditionally reside in the privileged domain while the user applications do not. Thus, to retain the benefits of separating the operating system from the user, we need a minimum of three hierarchical domains: kernel, supervisor, and user.

More general nonhierarchical domains could be useful in simplifying the design of a kernel-based system. Domain and capability machines fall into this class. Work on using a more general domain structure for kernel development is now in the research stage.

A process typically experiences a large number of calls to the kernel and supervisor, so we want mechanisms that ease the transfer of program control between domains. Entrance into the most privileged domain must, of course, be limited to well-defined entry points. A common means of limiting entrance is by using a system or supervisor fault or trap that transfers control to a known location. However, kernel simplicity and efficiency are improved if the hardware supports multiple entry points in a fashion more like a procedure call, so that each kernel function has a distinct entry point. The hardware should also support some form of argument validation (i.e., checking the validity of arguments passed by the application or supervisor to the kernel) and stack management for cross-domain calls. The Multics system employs a particularly elegant and efficient hierarchical domain architecture with domain-crossing hardware.[11]

Input/output mediation. I/O in most machines can take place in two fundamentally different ways. The simplest way, often called "programmed I/O," requires software to explicitly execute an I/O instruction to transfer each byte or word of information between an I/O device and a register or memory. We must therefore consider I/O devices as objects within the reference monitor framework; thus the kernel must control access to these devices. Typically, we would restrict the use of I/O instructions to the most privileged software domain (i.e., the kernel), and allow user and supervisor software to invoke kernel functions to perform I/O on their behalf.

A more complex architecture for I/O provides independent I/O processors that, once activated by the central processor, asynchronously transfer information between devices and memory. This transfer of information is specified by an explicit I/O program residing in memory or an implicit program (one built into the I/O processor) that is given parameters such as buffer and device addresses. The kernel must consider I/O programs in execution, or *I/O processes,* as subjects, and it must therefore control access to memory by I/O processors in the same manner as it controls access to memory by the CPU. As in programmed I/O, the conventional approach to handling this access control is for hardware to limit initiation of I/O processors (e.g., execution of a start I/O request) to the most privileged domain. I/O requests made by a user are in the form of kernel function calls. These calls cause a check of the I/O program or parameters to ensure that both the I/O devices and memory segments containing the I/O buffers are accessible to the user. The I/O processor itself, usually lacking multiple domains, typically works entirely in the kernel domain and uses physical memory addresses supplied by the kernel. Consequently, the kernel must often translate virtual addresses in the I/O program to physical addresses. Because of the sophisticated capabilities of some I/O processors, kernel checking of user-defined I/O programs can be a complex function.

Because of the complexity of handling I/O, a hardware architecture that allows direct user or supervisor domain access to I/O is desirable. Such an architecture would provide some form of descriptor to control access to the devices, in a manner similar to the use of memory descriptors. In addition, for the I/O processor to effectively operate outside the kernel by accessing virtual memory on behalf of the user, we need descriptor-controlled access to memory by the I/O processor. Such a capability is provided by the Scomp as discussed in the article by Fraim, which appears in this issue.

A careful concept of I/O operation should be part of the kernel development effort. We need to clearly distinguish between two device types: (1) external I/O involving devices such as terminals, local printers, and tape drives, which are effectively accessed by only one user at a time, and (2) internal I/O that includes devices such as disk drives and their storage media, which the kernel must manage because they can access information common to multiple users. For all removable I/O media on external devices capable of accessing information with varying access classes, we need a trusted labeling technique to ensure that the access class of the medium is correctly marked, or that some operator is in control of the access class of information accessible to the device. Labels can include, for example, nonforgeable banner sheets on printer output and operator interaction with the kernel for mounting tapes and removable disks.

Verification. Most kernel developments to date have been accompanied by some degree of formal verification. Some of the early promises of formal verification were overstated—verification has turned out to be more difficult than we expected.[12] Formal verification of a kernel involves problems of program correctness, and we are still quite a long way from being able to prove the correctness of a large computer program. Because formal verification technology has not fully matured, we need to understand its current capabilities before defining requirements for a major kernel development. Unrealistic expectations for verification can turn a practical development effort fully within the bounds of current technology into a research effort that could consume unlimited resources.

Many traditional nonmathematical methods such as structured design and testing can contribute to the overall confidence in the security of the kernel. With realistic goals, formal verification can enhance these traditional techniques and play a major and useful part in the kernel development process. Of the various stages of kernel development to which verification can be applied, the greatest degree of success has been obtained in specification verification. We have several techniques for verifying a formal specification against its model, and some of these, such as the flow analysis method mentioned earlier, have become almost routine. We can also verify correspondence between intermediate levels of specifications. However, even if there is no intent to complete a full mathematical proof, we still have the rigorous review, documentation, and kernel-development guidelines that most verification methodologies enforce. These alone will ensure a more secure and reliable system.

Although total confidence in the security of a system is not yet achievable, we can specify degrees of confidence in the security of different systems. The Department of Defense has recently promulgated a set of *Trusted Computer System Evaluation Criteria*[13] that define several distinct evaluation classes of progressively increasing confidence. These criteria explicitly recognize the value of following most of the security kernel design principles we have identified, yet they have widespread applicability to most types of systems, kernel-based or not.

Power line problems? No problem.

Analyze them with STABILINE® AC Voltage Monitors . . .

Truly portable multi-site disturbance analyzers – the *only* systems with detachable, microprocessor based Detector Modules which can be left at each site. Depending on whether 1- or 3-phase models are selected, Printer/Controllers can provide categorized formats of high/low voltages, surges, sags, impulses, common-mode noise, dropouts, power outages, d-c voltage, voltage and angle unbalances. Summaries include date, time, magnitude and duration of each category's 10 worst events and a list of the last 10 chronological events that qualify by threshold. Detector Modules can communicate with Printer/Controllers directly or over conventional telephone lines with suitable interfacing. Models available for 120, 240 or 480 VAC.

Solve them with STABILINE® Voltage Conditioning Equipment . . .

Once the problem has been pinpointed, condition aberrations that create malfunctions and erratic behavior: **Transient Voltage Suppressors** reduce the magnitude of (attenuate) voltage spikes or transients; **Automatic Voltage Regulators** deliver constant output voltage regardless of input line variations; **Computer Regulators** maintain properly conditioned voltage and attenuate common-mode noise and impulses; **Power Conditioners** regulate voltage, provide isolation for common-mode noise and reduce transverse-mode noise; **Transient Suppressors /Voltage Regulators** combine voltage correction and impulse attenuation; **Isolation Transformers** isolate equipment from common-mode noise generated by power consuming devices.

Write or call for STABILINE literature.

The Superior Electric Company

383 Middle Street / Bristol, Connecticut 06010 / (203) 582-9561 / Telex: 962-446

IN EUROPE: Superior Electric Nederland B.V.
The Hague, Netherlands / (070) 679590 / Telex: 31436 Supe nl

IN CANADA: The American Superior Electric Company, Ltd.
Toronto, Ontario / (416) 255-2318 / Telex: 06-967806

Reader Service Number 3

The security kernel design approach is the most promising methodology currently available that can provide both the internal security and the functional capabilities that many of today's computer systems need. This approach is based on a firm foundation and will support a wide range of commercial and governmental information protection policies. The kernel provides security controls that are effective against most internal attacks—including many that kernel designers never considered. Bugs of malicious software contained in applications, or even in the operating system, cannot cause unauthorized access to information.

The overall trend in hardware and software technology for computer systems is toward greater application of the principles and features applicable to the kernel approach. We have in fact recently seen the emergence of a security kernel in a commercial product (Honeywell Scomp). The required hardware and operating system technology is thus clearly within practical application. ■

References

1. B. W. Lampson, "Protection," *Proc. Fifth Princeton Symp. Information Sciences and Systems,* Mar. 1971, pp. 437-443.
2. L. Fraim, "Scomp: A Solution to the Multilevel Security Problem," *Computer*, Vol. 16, No. 7, July 1983.
3. C. Landwehr, "The Best Technologies for Computer Security," *Computer*, Vol. 16, No. 7, July 1983.
4. R. M. Needham and R. D. H. Walker, "The Cambridge CAP Computer and Its Protection System," *ACM Operating Systems Review,* Vol. II, No. 5; also in *Proc. Sixth Symp. Operating System Principles,* Nov. 1977, pp. 1-10.
5. D. E. Bell and L. J. LaPadula, "Computer Security Model: Unified Exposition and Multics Interpretation," tech. report ESD-TR-75-306, AD A023588, The Mitre Corporation, Bedford, Mass., June 1975.
6. K. G. Walter et al., "Structured Specification of a Security Kernel," *Proc. 1975 Int'l Conf. Reliable Software,* IEEE Cat. No. 75CH0940-7CSR, Los Angeles, Calif., Apr. 1975, pp. 285-293.
7. J. K. Millen, "Operating System Security Verification," *Case Studies in Mathematical Modeling*, W. E. Boyce, ed., Pitmann Publishing, Marshfield, Mass., 1981, pp. 335-386.
8. M. H. Cheheyl et al., "Verifying Security," *ACM Computing Surveys*, Vol. 13, No. 3, Sept. 1981, pp. 279-339.
9. B. J. Walker et al., "Specification and Verification of the UCLA Unix Security Kernel," *Comm. ACM,* Vol. 23, No. 2, Feb. 1980, pp. 118-131.
10. R. R. Schell, "The Structure of a Security Kernel for a Multiprocessor Microcomputer," *Computer*, Vol. 16, No. 7, July 1983.
11. M. D. Schroeder and J. H. Saltzer, "A Hardware Architecture for Implementing Protection Rings," *Comm. ACM*, Vol. 15, No. 3, Mar. 1972, pp. 157-170.
12. "Verification of Secure Software Systems," *Proc. 1980 Symp. Security and Privacy*, IEEE Cat. No. 80CH1522-2, Apr. 1980, pp. 157-166.
13. *Trusted Computer System Evaluation Criteria*, DoD Computer Security Center, Ft. Meade, Md., Jan. 1983.

Positions available in

COMPUTER SCIENCE & ROBOTICS

for graduates in

Mathematics
Computer Science
Electrical Engineering
Mechanical Engineering

Opportunities in programming and engineering:

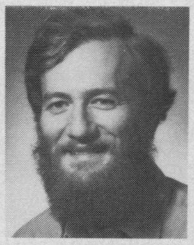
- VLSI design
- Supercomputer architectures
- Robotic systems and control
- Artificial intelligence
- Expert systems
- Intelligent management systems
- Image and speech processing
- Multiple large networks
- Personal computers
- Unix and VMS enhancements
- Network operating systems
- Multi-media message systems
- Programming environments
- Graphics applications

Send resumes to:
Director of Research Facilities
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213
(412) 578-2591

Carnegie-Mellon University

We are an equal opportunity/affirmative action employer.

Stanley R. Ames, Jr., is guest editor of this special issue on computer security technologies. His photo and biography appear on p. 12.



Morrie Gasser is a group leader at The Mitre Corporation in Bedford, Massachusetts. Since 1971, he has been involved in most aspects of computer security including hardware and software design and implementation, formal specification and verification, and computer security applications. His role at Mitre includes both in-house design and development as well as consulting to the Department of Defense on various research and development programs. Gasser received a BA in physics in 1969 from the University of Chicago.

Roger R. Schell is a colonel in the United States Air Force and is currently assigned as the deputy director of the Department of Defense Computer Security Center, Ft. Meade, Maryland. His interests include operating systems, software engineering, and computer security. From 1978 to 1981, he was associate professor of computer science at the Naval Postgraduate School in Monterey, California. Other experience includes serving as program manager and software engineer for several large military software developments, designing and implementing a dynamic reconfiguration for a commercial operating system, and introducing the security kernel technology.

Schell received a BS in electrical engineering from Montana State College, an MS in electrical engineering from Washington State University, and a PhD in computer science from the Massachusetts Institute of Technology.