Kernels for Safety?

Reprinted from T. Anderson, editor, *Safe and Secure Computing Systems*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989. (Proceedings of a Symposium held in Glasgow, UK, October 1986)

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA

Abstract

Secure systems are often built around a "security kernel"—a relatively small and simple component that guarantees the security of the overall system. In this paper we ask whether this approach can be used to ensure system properties other than security—in particular, we are interested in whether "safety" properties can be handled in this way.

Our conclusion is that kernelized system structures can provide rigorous guarantees that certain faults of commission will not occur. We give a more precise characterization in terms of the formal statement that can be asserted for a kernelized system and we outline an approach to system design that uses these insights and draws on experience with secure systems in order guarantee certain safety properties.

1 Introduction

Computer systems perform many critical tasks: tasks where certain kinds of failures cannot be tolerated. Failures are the result of *faults* and the prevention of failure therefore depends on eliminating faults, or on tolerating them safely. Techniques for fault elimination include requirements definition methodologies and languages, elaborate testing strategies, and the use of rigorous mathematical approaches to system design. These latter, which embrace formal specification and verification techniques, are among the most certain and effective, but are also the most difficult and costly to apply—their use demands more than usually skilled and well-trained designers and programmers, and the use of expensive and time-consuming tools.

Work on "secure" systems has evolved an approach to systems design based on the idea of a "security kernel": a small component of the total system whose correctness is sufficient to ensure the security of the system as a whole. The attraction is that the expensive and difficult techniques needed to guarantee correct behavior need only be applied to the relatively small security kernel—yet the benefit will apply to the complete system. It is interesting and potentially rewarding, therefore, to enquire whether the security kernel approach can be extended to apply to properties other than security.

In this paper, we suggest that the kernel approach is an appropriate way to achieve what we term *negative* properties: that is, properties which assert that "bad things don't happen"—as opposed to *positive* properties which assert that "good things do happen." Examples of negative properties include the requirements that a weapon should not become armed until it has been readied for firing, and that a message should not be released before it has been reviewed. We also provide a formal characterization of the properties that can be enforced by a kernelized system structure.

Previous investigations in this area include that of Froscher and Carroll [6], who have coined the term "process security" to describe a class of system requirements that may be regarded as extensions to the usual notion of security and may benefit from similar approaches to system design. However, our approach takes a more abstract view than that of Froscher and Carroll, since we characterize the appropriate requirements in terms of a certain class of "second-order" properties—rather than simply stretching the informal notion of "security." Members of the Honeywell SAT project independently developed an approach to enforcing an interpretation of "integrity" [2] that is in the spirit of the ideas proposed here, and have also extended a technique for specifying security requirements [9] along the same lines.

Leveson has posited the idea of a *safety kernel* [10] but, despite similarities in motivation and in the use of the term "kernel," her approach seems quite different to ours: a safety kernel seems to be more of a structuring concept than a mechanism for policy enforcement. We do not deny the merit in the idea of a safety kernel, we merely stress that our approach is different.

We begin by noting that the way in which hierarchical structure is exploited in a kernelized secure system is almost diametrically opposed the way it is used in more conventional system designs. We then characterize the class of requirements for which kernelized structure is appropriate in terms of "negative" properties and the need to avoid faults of commission. Next, we give a more precise characterization in terms of the formal statement that can be asserted for a kernelized system. Finally, we outline an approach to system design that uses these insights and draws on experience with secure systems in order to provide rigorous guarantees that certain faults of commission will not occur.

2 What Properties can a Kernel Enforce?

Hierarchical structure is generally advocated as a sound approach to systems design—especially for those systems that must satisfy exacting requirements. However, examination of cases where hierarchical structure is used to achieve such exacting requirements reveals that it is used for quite different, almost "opposite," purposes.

Techniques for both fault elimination and fault tolerance tend to induce a hierarchic system structure in which the most dependable service is achieved at the top. In complete contrast, work on systems which must satisfy exacting security requirements has led to an approach (based on the idea of a "security kernel") in which the most dependable component is located at the *bottom* of a hierarchical structure: the security kernel enforces security independently of the behavior of the software layers above it.

The question that prompted our investigation grew naturally from this observation of the two different forms of hierarchy. The question we asked is "what is it about security that makes the kernel approach so appropriate and effective"?

The essential characteristic of a security kernel is that it enforces security on the system as a whole without requiring the rest of the system to cooperate towards that end. We are interested in whether this approach can be extended to properties other than security. It is clear that any properties enforced by a kernel must necessarily concern, and be expressed in terms of, the operations provided by that kernel. However, the users of a system are not directly interested in its kernel at all; they are concerned with the behavior of the total system. Properties of a kernel are of interest only in so far as they contribute to the properties desired of the total system.

Thus, the systems for which a kernelized structure is appropriate are those in which properties of interest at the *system level* can be expressed in terms of functions provided at the *kernel level*. Of course, this requirement can always be achieved if we allow the kernel to be the whole system, but we only are interested in the case where the kernel is *much* smaller and simpler than the total system. In such a case, it is obvious that the kernel cannot, on its own, accomplish the overall system goal. Thus although the properties enforced by a kernel must be significant at the user level, they cannot amount to a statement of the overall purpose of the system.

Furthermore, it does not seem that a kernel can *enforce* conventional functional properties at all—since there is no guarantee that the rest of the system will use the kernel correctly. Suppose, for example, that the kernel provides certain functions that are basic to the accomplishment of the overall system goal. Great care may be lavished on the kernel and it may be asserted with confidence that it performs those functions correctly. Does this mean that the overall system can also be asserted to perform those functions correctly? Plainly not, for the non-kernel software may fail to invoke the relevant kernel functions correctly—it may even fail to invoke them at all, preferring instead to use its own, possibly incorrect, implementations of the relevant functions. Of course, we could suppose that really crucial functions are provided entirely within the kernel, but this seems to stretch the notion of "kernel" excessively. A kernel is surely a component that is used by other components, not a self-contained implementation of selected system-level functions.

It is worth stressing the key point here: no matter what "good" properties a kernel may possess, there can be no guarantee that the rest of the system will use the kernel correctly—and consequently no guarantee that the overall system will inherit the good properties of its kernel. We must therefore conclude that a kernelized system structure cannot be used to enforce "good behavior" on a total system. Instead, kernelized system structure seems better suited to the prevention of "bad behavior" through careful selection of the facilities that the kernel does *not* provide. These notions are developed in the next section.

2.1 Faults of Omission and of Commission

Systems responsible for performing critical tasks should work properly: they should ensure that good things happen and that bad things don't. Expressed more technically, they should avoid both faults of omission (failing to do what is required) and of commission (doing what is not required). Much effort has been devoted to the avoidance of faults of omission; faults of commission have received less attention.

Rather than looking at faults, it may be more helpful to focus on their absence. Let us term a statement that "good things happen" (i.e., there are no faults of omission) the assertion of a *positive* property; conversely, let us term a statement that "bad things do not happen" (i.e., there are no faults of commission) the assertion of a *negative* property.[1] Testing or verifying a system for the satisfaction of a positive property seems much more natural than doing so for a negative property. Consider a missile control system, for example. The positive property that the command to ready the missile always results in the warhead becoming armed can be ensured by thorough testing or, depending on the degree of assurance required, by formal program verification. It is not so clear that the same techniques can be used to deliver the same degree of assurance for the negative property that the warhead cannot become armed while the missile is lying on the deck. In the first case, we have only to ensure that a *certain* input leads to a desired output; in the second, we have to ensure that *no possible* set of inputs can lead to an undesired output.

If testing and verification seem better suited to positive than to negative properties, the situation seems reversed when kernelized system structures are considered. As we noted earlier, a kernel at the bottom of a hierarchy cannot force the layers above it to perform their tasks properly. Even if the kernel itself satisfies certain

[1] The literature on distributed systems uses the terms "safety" and "liveness" for classes of properties which have many similarities with our negative and positive classes.

positive properties, there is no guarantee that the rest of the system will use the kernel correctly. Thus, a kernelized system structure seems to provide no assistance in the satisfaction of positive properties.

Instead, a kernel can achieve influence over higher levels of the system through the facilities it does *not* provide—for if a kernel provides no mechanisms for achieving certain behaviors, and if no other mechanisms are available, then no layers above the kernel can achieve those behaviors. Thus, although it cannot enforce positive properties, kernelized design is admirably suited to guaranteeing *negative* properties. As an example, consider a missile control system built on a kernel that provides no paths for information, control, or data flow between any software component and the warhead arming mechanism, save that from the software component intended to perform the arming function. Then no errors in any but that one component can cause the warhead to be armed prematurely. In the next section we attempt a more formal characterization of properties of this sort.

2.2 First and Second-Order Properties

The terms "first" and "second-order" are taken from logic: second-order predicate calculus extends the first-order calculus by allowing quantification over functions as well as over individuals. Logic is relevant here because we are concerned with what properties are asserted (and in an ideal world, proved) about a system component.

The properties of many system components can be described by first-order assertions. Often these take the form of pre- and post-conditions given for each of its functions. Sometimes it is preferable to define how functions interact as, for example, in the case of a stack—where the first-order assertion

$$\mathrm{top}(\mathrm{push}(s, x)) = x$$

captures the interaction between the functions *top* and *push*. It seems that first-order assertions are the natural way of expressing positive properties: they describe what a component *does*.

The key fact about a security kernel, on the other hand, does not reside in the properties of its functions *individually*: a security kernel guarantees that a certain property (security) will hold no matter which, or how many, of its functions are invoked, or in what order. The formal statement of this property requires a second-order formula—in particular, one of the form

$$\forall \alpha \in op^* : P(\alpha), \tag{1}$$

where op^* denotes the set of all sequences of invocations of functions provided by the kernel and $P(\cdot)$ is a predicate over the input/output behavior of that set.[2]

[2]Those interested in the precise $P(\cdot)$ that describes multilevel security are referred to the papers that describe what has become known as the "SRI Security Model" [4, 5, 8, 7, 14]—essentially, it

The second-order formula (1) expresses the following important property: provided every operation that can be performed by non-kernel software ultimately comes down to a sequence of calls on the kernel interface (i.e., functions in the set *op*),[3] and provided the kernel protects itself from modification,[4] the property $P(\cdot)$ will be satisfied by the resulting system, *no matter what the behavior of the non-kernel software.*

We claim that the second-order assertion (1) is the *sine qua non* of the approach to hierarchical structuring exemplified by a security kernel. A kernel cannot force the rest of the system to use its functions correctly; the only leverage possessed by a kernel is that every action performed by the rest of the system eventually comes down to a sequence of calls on the kernel interface (ordinary hardware instructions are considered part of the kernel interface). Since the kernel cannot influence which of its functions will be invoked by the rest of the system, nor the order in which they will invoked, the property guaranteed by the kernel must be one that holds over all possible sequences of invocations of its functions—and this is precisely the property stated by the assertion (1).

Assertion (1) is somewhat abstract in that it uses a predicate over the input/output behavior of arbitrary function sequences. A more concrete system model would employ a notion of system *state* and would reduce (1) to an assertion that each function of the kernel interface leaves a certain property of the system state *invariant.*[5] This reduced form can be regarded as a first-order assertion on each function separately, and proved accordingly—thus the use of the more powerful second-order logical formalism does not complicate the process of verification. In fact, many of the properties required of a kernel may be most naturally specified directly in terms of an invariant on the system state.

We can now see that two requirements must be satisfied by a system in order for it to be a candidate for the security-kernel approach to systems design:

1. The properties of interest at the system level must be present at the kernel level, and

2. Those properties must be expressed by a second-order assertion of the form (1) (or, as a special case, in the form of an invariant on the system state).

The classical multilevel security property has exactly these characteristics—which is why security kernels have proved an appropriate approach to the design of such systems. Not all security problems are amenable to this approach, however. The

states that inputs from users operating at high classifications must have no effect on the behavior perceived by less highly cleared users.

[3]This assumption is generally referred to as *completeness* in the security literature.

[4]This assumption is often referred to as *isolation* in the security literature.

[5]In the case of security, this reduction has become known as "unwinding" [8].

problems of aggregation and inference cannot be handled by a security kernel because the properties of interest cannot be expressed at the level of kernel operations (i.e., the first of the two requirements listed above is not satisfied).

The second of the requirements above is a more precise characterization of the association, developed intuitively in the previous sections, between kernelized systems a negative properties: it simply seems to be the case that, whereas a first-order formula provides the natural expression of a positive property, so the second-order formula (1) seems naturally suited to the assertion of negative properties. In the next section, we develop practical consequences of these insights.

3 Implications for System Design

Work on the construction of "secure" systems—systems which guarantee not to disclose classified information to those unauthorized to view it—has led to the idea of structuring systems around a "security kernel." Crudely, the idea is that the security kernel should contain everything that can affect security, and should exclude everything else. The task of guaranteeing security is then reduced to showing that non-kernel components cannot affect security, and that all kernel components preserve it. The decomposition of a secure system into kernel and non-kernel components is orthogonal to any functional decomposition that may also be present; the security kernel generally has much of the character of an operating system nucleus and serves to support and limit the behavior of non-kernel components. More recently, the partitioning of a secure system into simply kernel and non-kernel components has come to be seen as excessively coarse [1, 11]. The maintenance of security is now seen to require a symbiosis between a security kernel and a number of non-kernel "trusted functions"—the whole assemblage of security-critical components being called a "Trusted Computing Base" (TCB) [3].

The approach to TCB design that we espouse [15] reduces the role of the security kernel to that of a *domain separation mechanism* and we use the term *separation kernel* to describe this more limited form of kernel. The purpose of the separation kernel is to create encapsulated *domains* and controlled paths for inter-domain communication. The significant point about domains is that one cannot influence the behavior of another, except through one of the explicit inter-domain communications channels. These channels are provided by the kernel, which can therefore control interaction between domains.

The separation kernel is only the first level in our TCB; the other level comprises the trusted *resource managers*. In a secure system, these resource managers are trusted to provide multilevel-secure access to the resources which they encapsulate. Notice that both the separation kernel and the resource managers enforce negative properties. In the case of the kernel it is the property "this domain is not allowed to influence that domain," while in the case of a multilevel secure resource manager it is the "noninterference" definition of security [5, 7, 14].

We believe this TCB structure is well suited to enforcing negative properties other than security, and thereby to preventing certain types of errors of commission. The limited inter-domain communications channels provided by a separation kernel minimize the possibility that errors in one domain will contaminate the behavior of others. In addition, the separation kernel ensures that the encapsulation provided by resource mangers is total—so that a resource cannot be accessed or modified except by requests sent to its resource manager. Those resource managers may then enforce certain negative properties of their own (typically the maintenance of invariants on their internal state).

The simplest property that may be specified for a separation kernel is one that says "this domain is not allowed to influence that domain." Even this simple property has useful applications: we can ensure that a weapon's guidance system, say, cannot arm its warhead.

A more interesting application concerns the correct *sequencing* of operations. Many safety and security issues can be reduced to the need for correct sequencing: for example, a missile launch sequence "ready, aim, fire," and the rules concerning the release of messages (a message must be reviewed by a release officer before it is allowed out). Simple, static, sequencing rules can be built directly into a separation kernel as part of the "wiring diagram" of inter-domain communication paths. For example, if the only input channel to the "fire" domain is connected to an output channel from the "aim" domain, then the weapon must be aimed before it is fired. Of course, this simple solution admits the possibility that the "aim" domain may enable the "fire" domain without having performed the aiming function correctly, but at least we know that no other malfunction can cause the weapon to be fired without being aimed. A specification of how domains should be "wired up" is referred to as a "channel control policy" [12] and a method for formally verifying that a separation kernel enforces such a policy is described in [13].

More sophisticated sequencing properties can be enforced by a separation kernel which monitors the "status" of information that moves over its communication paths. By the *status* of information, we include not just its "syntactic" type (e.g., "text"), but also its "semantic" state (e.g., "not yet given a security classification", or "classified, but not yet reviewed"). The kernel can view each domain as a "status transformer" and by knowing the status of information required by one domain and the status of the output produced by another, it can consult a "policy base" in order to determine whether the output of the latter is acceptable input to the former. For example, in a process control system with a man-in-the-loop, the domains responsible for driving the actuators would be allowed to receive as input only those messages with status "approved," and the only domain capable of generating outputs with this status would be that which encapsulates the human operator whose approval is required. A special advantage of this scheme is that it extends easily to distributed systems. Workers at Honeywell have demonstrated the power and flexibility of this technique by using it to solve a number of difficult problems, including one related to integrity [2], and have also used the noninterference assertions of Goguen and Meseguer [8] to generate formal specifications for these policies [9].

In the TCB structure described here, the chief responsibility of the separation kernel is the control and limitation of inter-domain communications. The responsibility of the resource managers will be to maintain the safe condition of the resources which they encapsulate. A resource manager responsible for some item of equipment, for example, will maintain the invariant that the equipment is within its safe operating envelope.

4 Conclusions

We have argued that a system kernel can influence the behavior of the whole system through the selection of functions it does *not* provide. By denying the ability to achieve certain behaviors, the kernel can prevent certain faults of commission. An abstract characterization of the class of behaviors that can be enforced in this way is given by a simple second-order formula (1). Special cases of this formula are noninterference specifications [8, 9] and invariants on (parts of) the system state. A Trusted Computing Base (TCB) comprising a separation kernel and zero or more resource managers is appropriate for both cases. The separation kernel enforces the noninterference requirement (causing otherwise isolated "domains" to be "wired up" appropriately), while the resource managers maintain certain properties invariant. Examples have been given of a number of what might reasonably be called "safety" properties that can be specified and enforced by these techniques.

5 Acknowledgements

The ideas presented in this paper were sharpened considerably in discussions with Michael Melliar-Smith.

References

[1] S.R. Ames Jr. Security kernels: a solution or a problem? In *Proceedings of the Symposium on Security and Privacy*, pages 141–150, Oakland, CA, April 1981. IEEE Computer Society.

[2] W.E. Boebert and R.Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings 8th DoD/NBS Computer Security Initiative Conference*, pages 18–27, Gaithersburg, MD, September 1985.

[3] *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, December 1985. DOD 5200.28-STD (supersedes CSC-STD-001-83).

[4] R.J. Feiertag. A technique for proving specifications are multilevel secure. Technical Report CSL109, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1980.

[5] R.J. Feiertag, K.N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Proc. 6th ACM Symposium on Operating System Principles*, pages 57–65, November 1977.

[6] J.N. Froscher and J.M. Carroll. Security requirements for navy embedded computers. NRL Memorandum Report 5425, Naval Research Laboratory, September 1984.

[7] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982. IEEE Computer Society.

[8] J.A. Goguen and J. Meseguer. Inference control and unwinding. In *Proceedings of the Symposium on Security and Privacy*, pages 75–86, Oakland, CA, April 1984. IEEE Computer Society.

[9] J. Haigh and W. Young. Extending the non-interference model of MLS for SAT. In *Proceedings of the Symposium on Security and Privacy*, pages 232–239, Oakland, CA, April 1986. IEEE Computer Society.

[10] N.G. Leveson et al. Design for safe software. In *Proc. AIAA 21st Aerospace Sciences Meeting*, Reno NV, January 1983. American Institute of Aeronautics and Astronautics.

[11] John Rushby. The design and verification of secure systems. In *Proc. 8th ACM Symposium on Operating System Principles*, pages 12–21, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[12] John Rushby. Verification of secure systems. Technical Report 166, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, August 1981.

[13] John Rushby. Proof of Separability—a verification technique for a class of security kernels. In *Proc. 5th International Symposium on Programming*, pages 352–367, Turin, Italy, April 1982. Springer-Verlag Lecture Notes in Computer Science, Vol. 137.

[14] John Rushby. The security model of Enhanced HDM. In *Proceedings 7th DoD/NBS Computer Security Initiative Conference*, pages 120–136, Gaithersburg, MD, September 1984.

[15] John Rushby. A trusted computing base for embedded systems. In *Proceedings 7th DoD/NBS Computer Security Initiative Conference*, pages 294–311, Gaithersburg, MD, September 1984.