Design and Verification of Secure Systems

Reprint of a paper presented at the 8th ACM Symposium on Operating System Principles, Pacific Grove, California, 14–16 December 1981. (ACM Operating Systems Review Vol. 15 No. 5 pp. 12-21)

John Rushby*
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

This paper reviews some of the difficulties that arise in the verification of kernelized secure systems and suggests new techniques for their resolution.

It is proposed that secure systems should be conceived as distributed systems in which security is achieved partly through the physical separation of their individual components and partly through the mediation of trusted functions performed within some of those components. The purpose of a security kernel is simply to allow such a 'distributed' system to actually run within a single processor; policy enforcement is not the concern of a security kernel.

This approach decouples verification of components which perform trusted functions from verification of the security kernel. This latter task may be accomplished by a new verification technique called 'proof of separability' which explicitly addresses the security relevant aspects of interrupt handling and other issues ignored by present methods.

*This work was performed while the author was with the Computing Laboratory, University of Newcastle upon Tyne, England, and was sponsored by (what was then) the Royal Signals Radar Establishment.

Introduction

A formally verified security kernel is widely considered to offer the most promising basis for the construction of truly secure computer systems, at least in the short term. A number of kernelized systems have been constructed [12,19,25] and various models of security have been formulated to serve as the basis for their verification [6, 9, 28].

Despite the enthusiasm for this approach, there remain certain difficulties and problems in its application (see, for example [1]). I shall expand on these later, but briefly they include the difficulty of verifying the 'trusted processes' that seem necessary in most applications, concern about the extent to which current techniques verify the implementation of the kernel (as opposed to its specification), and doubts about whether present security models really capture the essential characteristics of a security kernel with sufficient accuracy to provide a sound technical basis for their verification. Also, current approaches to kernel design and verification developed out of concern for the problem of providing multilevel secure operation on general-purpose multi-user systems—whereas many of the present-day applications which require some form of guaranteed security are special-purpose, single-function systems [5, 11, 13, 24, 33] whose security requirements are somewhat different to those enshrined in the multilevel models. Attempts to support these applications on a conventional kernel have led to systems of considerable complexity whose verification presents difficulties that are quite at variance with the evident simplicity of the task which the system is intended to perform [2].

The purpose of this paper is to present a new approach (or, rather, a re-working of some old approaches [3,26,27]) to the design and verification of secure systems and to argue that the problems of conventional kernelized systems are thereby avoided or overcome.

The presentation is divided into four sections. In the first, I shall argue that the problems with conventional systems have their roots in the use of a security kernel which attempts to impose a single security policy over the whole system. The second section will propose that ***distributed*** systems avoid many of these difficulties and provide a more appropriate conceptual base for the design of secure systems. In such a system, the subjects of the security policy are assigned to private and physically isolated single-user machines and are able to communicate with each other and to access shared resources only through the mediation of specialised (and verified) 'trusted components' that reside in similarly isolated and dedicated machines. The overall security of such a distributed system rests partly on the physical separation of its components and partly on the critical functions performed by the trusted components. The concrete nature of the services provided by these components, and the limited interaction between them, enables their security properties to be specified and verified comparatively easily, and by existing techniques.

Next, in section 3, I shall argue that a conceptually distributed system can be supported on a single processor, while retaining its security properties, if a type of security kernel which I call a 'separation kernel' is used to simulate the distributed environment. There is absolutely no interaction between the properties required of a kernel of this type and the security properties required of the system components which it supports.

Finally, in Section 4, I shall outline a precise specification of the role of a separation kernel and sketch an appropriate method of verification which I call 'Proof of Separability' and which is developed formally in a companion paper to this [31]. The mathematical model which underlies this method of verification explicitly addresses the interpretive character of a security kernel and provides a sound formal basis for verifying the security relevant aspects of interrupt handling and other issues concerning the flow of control which are ignored by present methods.

1 The Problem of Trusted Processes

The primary motivation for the use of a security kernel is the desire to isolate and localise all 'security critical' software in one place—the kernel. Then, if the kernel can be proven 'secure' in some appropriate sense, all non-kernel software becomes irrelevant to the security of the system. Security kernels differ in the extent to which they are cognizant of the overall security policy of the system. Some kernels (for example, that of UCLA Secure UNIX [25]) have the character of a sophisticated protection mechanism and guarantee that no object supported by the kernel may be accessed in any way unless its recorded 'protection data' explicitly permits that type of access. The task of setting up the protection data so that it enforces some overall security policy is delegated to a 'policy manager' outside the kernel. The limitation of this approach is that it is concerned only to protect the physical representations of information, rather than information itself. Thus it does not control the 'leakage' of information through covert signalling paths [15, 17], nor is the notion of such 'information flow' expressible in the model [28, 32] which underlies the verification of these kernels.

In military applications, *all* unauthorized flow of information, whether due to direct access or indirect leakage, is unacceptable and, in consequence, security kernels intended for these applications must not only enforce the security policy of the system on all non-kernel software, but must also adhere to it themselves, in order that their own internal variables may not become a channel for insecure information flow [17, 20]. This implies that the kernel must enforce and obey a single, system-wide security policy. But once this approach is adopted, it is soon discovered that certain system functions cannot be accommodated within its discipline. A line-printer spooler provides a simple example of such a function: if the spooler and its spool files are at the highest security classification, then users of more lowly classification cannot inspect their own spool files—even for the innocent purpose of discovering the progress of their jobs. For this reason, it is usual for spool files to be classified at the level of their owners while the spooler continues to run at the highest level so that it may read spool files of all classifications. But then the spooler cannot delete spool files after their contents have been printed—for such action conflicts with the (kernel enforced) *-property [6] of multilevel security. In order to provide an acceptable user interface, while avoiding the proliferation of used spool files, it seems necessary that the spooler should become a 'trusted process' and be allowed to violate the *-property.

In real systems there are many functions which require the privileges of trusted processes in order to evade or override the security controls normally enforced by the kernel. In KSOS, for example, the trusted processes contain

> "support software to aid the day-to-day operation of the system (e.g., secure spoolers for line printer output, dump/restore programs, portions of the interface to a packet switched communications network etc.)." [7, page 365]

Once trusted processes are admitted to the system, however, the kernel is no longer the sole arbiter of security; it is necessary to be sure that the special privileges granted to trusted processes are not abused by those processes and may not be usurped by other, untrusted, processes. In order to guarantee security, therefore, we must verify the whole of the 'trusted computing base'—that is, the combination of kernel and trusted processes. The difficulty is that existing formal models do not provide a basis for the verification of this combination: we do not know what it is that we have to prove! Landwehr, for example, observes:

> "...in the final version of their model, Bell and LaPadula did include trusted processes. What is not included in their exposition is a technique for establishing when a process may be trusted." [16, page 46]

In the absence of any precise formulation of the role of trusted processes within a model of secure system behaviour, and in the absence of any formal understanding of how properties proved of trusted processes combine with those proved of a security kernel in order to establish the security of the complete system, there is no real justification for speaking of the 'verification' of the security of such systems at all.

The existence of trusted processes within kernelized systems and the attendant difficulties of verifying the security of those systems should not be attributed to deficiencies in the design of individual kernels, however. Rather:

> "to a large extent they [trusted processes] represent a mismatch between the idealizations of the multilevel security policy and the practical needs of a real user environment." [7, page 365]

The true roots of the difficulties caused by trusted processes are not to be found in those processes themselves, nor in the functions which they perform, but in the conception that a security kernel should act as a centralized agent for the enforcement of a uniform system-wide security policy. Even within a system which is intended to enforce a single security policy at its external interface, the rules and restrictions that govern the behaviour of its own components cannot simply be that overall policy in microcosm, but must be particular to the function of each component and to its individual role within the larger system. The properties required of a secure line-printer spooler, for example, depend as much on the fact that it ***is*** a line-printer spooler as on the security policy that is to be enforced. We should seek a system structure that allows each component to make its own contribution to the security of the overall system and that treats all contributions equally—as befits the 'weakest link' nature of security. We should not elevate the security requirements particular to one class of components to a special status and impose them system-wide at whatever inconvenience to components with different requirements. The truth of this proposition becomes self-evident when we consider some of the specialised applications of secure systems. The ACCAT Guard provides a good example [33].

The Guard is basically a facility for the exchange of messages between a highly classified system and a more lowly one. Messages from the LOW system to the HIGH one are allowed through the Guard without hindrance, but messages from HIGH to LOW must be displayed to a human 'Security Watch Officer' who has to decide whether they may be declassified to the level of the LOW system and then allowed through. Notice that the Guard supports information flow between the LOW and HIGH systems in ***both*** directions and has to enforce ***different*** security requirements on each. It is plainly inappropriate, therefore, to base its construction on a security kernel that enforces the requirements for just one direction of transfer—yet this is exactly what has been done. The Guard is based on the KSOS kernel—which enforces a multilevel security policy that permits information flow in only the LOW to HIGH direction. Consequently, the HIGH to LOW transfers have to be accomplished by trusted processes whose purpose is to get round the fundamental security principle of the KSOS kernel. It is not clear how the use of this kernel has contributed to the overall security or verifiability of the Guard and it is certainly no surprise to learn that:

> "Verification of the trusted processes to be used in the Guard has consumed far more resources than originally planned." [16, page 46]

2 Security and Distributed Systems

The combination of a security kernel and trusted processes is hard to understand and even harder to verify because it does not represent a separation of concerns but a confusion of the same: neither member of the combination is independent of the properties of the other. If we are to gain a clearer understanding of the nature of secure systems, and a more compelling basis for their verification, then we should attempt to separate the properties required of a security kernel from the issues that give rise to trusted processes.

A very simple and natural—in fact obvious—model for a computer system where security does not rely upon a central mechanism (such as a security kernel) is a functionally distributed system: one whose various functions are provided by specialised individual subsystems which are physically separated from each other and provided with only limited channels for communication with one another. Once such a system structure is adopted, a lot of security problems just vanish and others are considerably simplified.

Consider, for example, the problem of providing a multilevel secure service to a number of users in which files are to be the only medium of information flow between users of different security classifications. We can imagine an idealized system in which each user is given his own private, physically isolated, single-user machine and a dedicated communication line to a common, shared file-server. The only component of this system that needs to be trusted is the file-server. Provided that single component adheres to and enforces the multilevel security policy, the security of the rest of the system follows from the physical separation of its components and the absence of direct communications paths between users of different classifications.

Now consider the file-server in more detail. It is a system dedicated to a single purpose: it supports no user programming and needs no operating system since it runs just one program—the file-server program. In order to guarantee the security of the whole system, all we need to do is to verify that single program with respect to an appropriate specification of its security requirements. It turns out that the role of a multilevel secure file-server matches the security model developed at SRI [9] (which is more than can be said of a security kernel—a point I shall return to later) and this model therefore provides both a specification for the security requirements of the file-server and the justification for its verification by the method of 'information flow analysis' [8, 20, 21].

We can add further shared resources to the system in just the same way as the file-server. A central printing facility, for example, can be provided by a self-contained printer-server connected to each single-user machine (and probably the file-server also) by additional, dedicated communication lines. The printer-server must obviously satisfy some security requirements. It must, for example, print the correct security classification of each job on its header page and must not print parts of one job within another, nor feed inputs from one user back to another, and so on. Furthermore, the printer-server may need to co-operate with the file-server and may require services from the file-server that are different from those provided to ordinary users (for example, the ability to delete spool files of all security classifications). Whatever the full set of requirements for a secure printer-server are, they must be, at least in part, specific to its particular function; we cannot expect the security requirements of so specialised a task to be completely expressed by, or even to be totally consistent with, some general set of properties such as the ss- and *-properties of multilevel security [6]—even though enforcement of multilevel security is the overall goal.

We are, however, in a much better position to tackle the important problem of deciding just what *are* the requirements for a secure printing service when all responsibility for this service is completely isolated and exposed within a self-contained component, than when it is divided, uneasily and obscurely, between a trusted process and a security kernel.

A real system will contain more security-critical functions than just file and printer-servers. There must, for example, be some additional mechanism to authenticate the identities of users as they log in to the single-user machines and to inform the file and printer-servers of the security classifications associated with each user.

I contend that the security properties required of these and other critical services can best be studied if they, too, are isolated as separate, specialised components within a distributed system. The task of the system designer is then to identify and formulate the security properties that must be required of each component individually so that, in combination, they enforce the security policy required of the system overall.

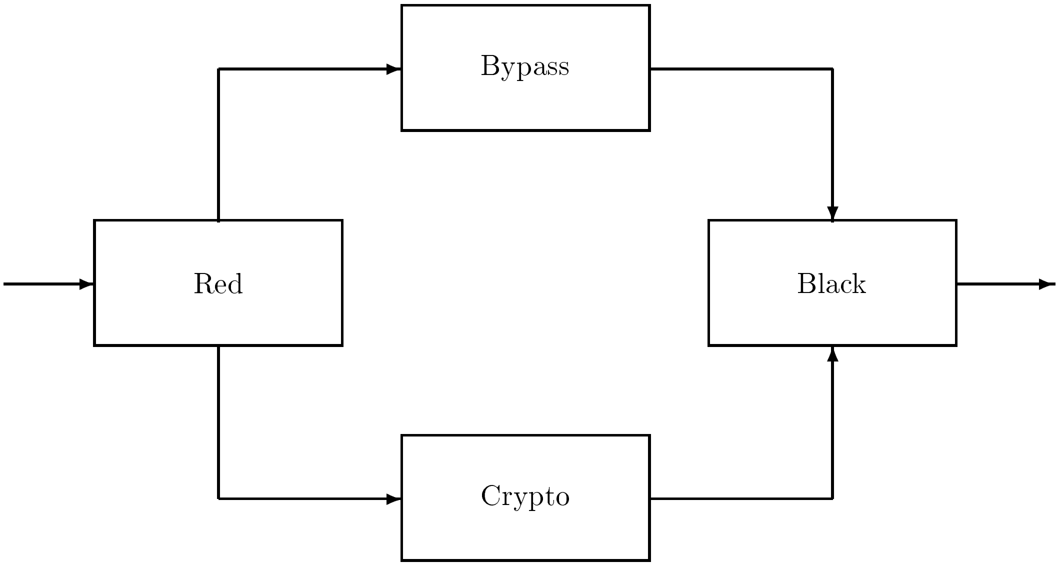
Of course, sceptics will point out that this is a formidable task: the components of the system interact and cannot be studied independently of each other. The printer-server, for example, requires special services of the file-server and both of these components depend upon information provided by the authentication mechanism. But the difficulties that appear formidable here are no less so in a conventional, kernelized system: the same functions and the same interactions must be present there also—and will be no less significant, merely less visible. Furthermore, the interactions in a distributed system are between its critical components. These components have concrete tasks to perform and their interactions can also be specified concretely: we can state precisely what the special services are that the printer-server requires of the file-server and we can satisfy ourselves that the ramifications of these special services are fully understood. This is quite different to granting the line printer spooler of a kernelized system a dispensation to flout the *-property.

Although I have been using a general-purpose multi-user system as a familiar example to introduce the idea, political and economic considerations generally dictate that secure general-purpose systems should emulate some existing system—and this hampers the adoption of a radically different implementation technique. Special-purpose, single-function systems are not so constrained—and are more able and more likely, therefore, to take advantage of a 'distributed' approach to security. A design for a type of 'secure network front end' (SNFE) will serve as an illustration.

A SNFE is a device that is interposed between host machines and a network in order to provide end-to-end encryption around the network. Some of the general design issues for such a device are discussed by Auerbach [4] and a particular design is described by Barnes [5]. Basically, the issues are as follows. As well as a cryptographic device (a 'crypto') the SNFE must certainly contain components for handling the protocols, message buffering and so on required at its interfaces with the communications lines to the host on one side and the network on the other. We can call the component on the host side the 'red' component and that on the network side the 'black' component. (This terminology stems from cryptological usages.) Packets of cleartext data from the host are received by the red component and passed to the crypto from where they travel, in encrypted form, to the black component for transmission over the network. In order to allow for red-black co-operation (essentially, the exchange of packet headers), a second, unencrypted channel (the 'cleartext bypass') must also connect the red and black components.

The security requirement of the system is that user data from the host must not reach the network in cleartext form. It is therefore necessary to be sure that the red component does not use the cleartext bypass to send user data directly to the black component. The software in the red component is considered too large and complex to allow its verification and so a 'censor' is inserted into the bypass to perform rigid procedural checks on the traffic passing through—to check that it has the appearance of legitimate protocol exchanges, rather than raw cleartext. A fairly simple censor can reduce the bandwidth available for illicit communication over the bypass to an acceptable level.

Observe that the crucial issue here is not *whether* red and black can communicate, but *what channels* are available for that communication: the channels via the censor and the crypto are allowed, but there must be no others. It is not clear how this requirement could be expressed in terms of the models that underly current conceptions of a security kernel but it is easily formulated and understood in the context of a distributed system design: the four components of the system are housed in separate, isolated boxes and connected by just the communications lines shown in the diagram. The only software which performs a security critical task in this design is that of the censor (the crypto is a trusted physical device); security is otherwise achieved by the physical distribution of the components and the physically limited communications provided between them.



3 Re-introducing the Security Kernel

So far I have argued that distributed systems offer a natural basis for the design of computer systems that must satisfy certain security requirements. Recent hardware developments make it feasible, for certain applications, to implement such designs directly—that is, as physically distributed systems composed of independent processors connected by external communications lines.

More commonly, however, and especially when the number of components in the distributed design is large relative to the overall scale of the system, it will be more cost-effective to implement the entire system on a single processor. In this case, the security characteristics of the distributed system must be provided by logical rather than physical mechanisms and this can be accomplished by re-introducing the concept of a security kernel, but in a different guise to that seen previously.

The overall security of a distributed system rests partly on the physical separation of its components and partly on the critical functions performed by some of those components. The role which I propose for a security kernel is simply that it should re-create, within a single shared machine, an environment which supports the various components of the system, and provides the communications channels between them, in such a way that individual components of the system ***cannot distinguish*** this shared environment from a physically distributed one. If this can be achieved, then surely the shared implementation retains all the security properties of a truly distributed system. Observe that such a kernel knows nothing of the security policy enforced by the system—that responsibility remains embedded in the critical components. And notice, too, that those critical components require no special privileges of the kernel; we have completely decoupled the properties required of the security kernel from those concerned with the larger questions of the system's overall purpose and policy.

In an ideal, physically distributed implementation, each component of the system runs on its own private and physically isolated machine. The task of a security kernel, therefore, is to provide an isolated 'Virtual Machine' (VM) for each component and to handle communications between these virtual machines. A kernel of this form is obviously very similar to a 'Virtual Machine Monitor' (VMM): that is, a system which provides each of its users with a separate, simulated copy of its hardware base (VM/370 is, perhaps, the best known example of such a system). It is widely recognised that VMMs provide a suitable basis for the construction of secure systems and at least two systems have been constructed along these lines [12, 26]. However, the type of kernel which I am proposing differs from a VMM in that there is no requirement for it to provide VMs which are exact copies of the base hardware (or even for all the VMs to be alike)—but there is a requirement for it to provide communications channels between some of its VMs. In order to avoid confusion with established terminology, I shall call this new type of security kernel a 'separation kernel' and I shall speak of the VMs which it supports as 'regimes.'

The next step is to deduce a precise statement of the security properties required of a separation kernel and to develop a technique for verifying these properties. Before doing so, however, it seems best to assist the reader's intuition and to provide some motivation by outlining the design of a particular separation kernel.

An Example

The separation kernel concerned is an operational one known as the 'Secure User Environment' (SUE). It runs on a PDP-11/34 and was designed and constructed by T4 Division of the Royal Signals and Radar Establishment at Malvern, England, in order to support applications similar to the SNFE described earlier. One of the chief design aims of the SUE was that it should be minimally small and very simple [5]. (The SDC Communications Kernel [11] is a similar system, though rather more complex.)

Because the SUE is only required to provide a fixed (and small) number of regimes, each of which executes a fixed (and small) program, there is no need for it to support paging or virtual memory management as found in the kernels of general-purpose systems such as KVM/370 [12]. Instead, a much simpler memory-resident system is possible in which each regime is permanently allocated to a fixed partition of real memory while the SUE itself occupies another fixed partition. The SUE manipulates the memory management features of the PDP-11/34 in order to arrange for its own protection and the mutual isolation of its regimes.

In order to further reduce its size and simplify its design, the SUE performs no scheduling functions. Regimes are given control on a round-robin basis and execute until they suspend voluntarily (via a SWAP call to the SUE). Because the whole system is dedicated to a single function, 'denial of service' is not a security problem (although it is clearly a reliability issue).

Input/output via Direct Memory Access (DMA) poses a security threat on most machines (including PDP-11s) since it uses absolute addresses and thereby evades the protection of the memory management hardware. For this reason, conventional kernels must handle or mediate all I/O operations and this is a source of significant complexity in their design. The SUE adopts a far more ruthless approach: DMA is permanently excluded from the system. (The efficiency problems this might seem to cause are overcome by the use of special-purpose hardware [18].) With DMA excluded from the system, almost all responsibility for I/O can be removed from the SUE since the memory management of a PDP-11 allows device registers to be protected just like ordinary memory locations. Each device supported by the system is permanently and exclusively allocated to a fixed regime and its device registers are located in the address space of that regime. Responsibility for each device then rests with the regime which controls its device registers. The only responsibility of the SUE with respect to I/O activity is to field interrupts (since the hardware vectors these through kernel address space) and pass them on to the appropriate regime for handling. Return from interrupts similarly requires minor assistance from the SUE.

Apart from the provision of the communications channels that are required between certain regimes, this description has summaried just about the whole of the SUE. Readers will appreciate that, in comparison with a conventional security kernel, the SUE is indeed small and simple. (It occupies about 5K words, including all stack and data space.) What we seek now is a verification technique that exploits this simplicity in order to provide perspicuous and compelling evidence of the SUE's security.

4 Verification

The task of a separation kernel is to create an environment which is indistinguishable from that provided by a physically distributed system: it must appear as if each regime is a separate, isolated machine and that information can only flow from one machine to another along known external communications lines. One of the properties we must prove of a separation kernel, therefore, is that there are no channels for information flow between regimes other than those explicitly provided. In the case of the SNFE described earlier, for example, there must be no direct channels between the red and black regimes—although the channels via the crypto and the censor are quite legitimate. By allowing certain channels and demanding the absence of all others, we create a rather difficult verification problem. It would be much easier to demand the absence of *all* channels—that would correspond to a policy of isolation and seems a more reasonable candidate for verification. Analogy with a physically distributed system suggests how the original problem can be simplified in this way: *if we cut the communication channels that are allowed*, then, provided there are no illicit channels present, *the components of the system will become completely isolated from one another*. It now remains to discover how to 'cut' communication lines that are not physical wires but properties of the kernel software.

The solution to this problem is easily seen once we consider how communication is actually accomplished in software—by the use of shared objects. If regimes A and B have a communication channel between them, then there must, at bottom, be some shared object, say X, which the sender can write and the receiver can read. If we now replace all of A's references to X by references to a new object, X1, and all of B's references to X by references to another new object, X2, then this is equivalent to 'cutting' the communication channel represented by X, with X1 and X2 taking the parts of the two 'ends' produced by the cut. If, following this 'cutting' of the 'X channel,' we are able to demonstrate that the A and B regimes have become isolated, then it follows that this was the *only* channel between them.

This is an indirect argument and may appear specious to some: we prove a property (isolation) of one system (that with its 'wires cut') and infer another property (absence of illicit channels) of a different system. However, if the differences between the two systems are of the very limited, controlled form that I have described (involving only the 'aliasing' of certain names), so that the consequences of the differences between them may be understood *completely*, then, surely, the technique is sound. (For more extended discussion, and an example of the application of the technique, see [30].)

We now need a method for proving that a separation kernel (with its 'wires cut') enforces isolation on its regimes: we must prove the total absence of any information flow from one regime to another. The technique which has been used to verify secure information flow in kernels constructed by the Mitre Corporation [20] and in KSOS [7, 10], and which seems to be widely accepted, is known as 'information flow analysis' (IFA) [21]—sometimes also called 'security flow analysis.' It might be thought that this will also provide a satisfactory technique for verifying a separation kernel. But this is not so.

One reason for this is that IFA cannot verify some of the machine-level manipulations that must be performed by a separation kernel—the SWAP operation provides a simple example.

Consider a separation kernel supporting just two regimes, identified as RED and BLACK. When the RED regime is executing, it may relinquish the CPU by performing a SWAP operation. The effects of this operation must include the saving of the current contents of the general registers in a RED save area, and their reloading with values from a BLACK save area. Verification by IFA requires that operations invoked by RED may only access RED values—but it is evident that the SWAP operation ***must*** access ***both*** RED ***and*** BLACK values. It follows that IFA cannot verify the security of a SWAP operation, even though it is manifestly secure (see [30] for more extended discussion and some worked examples). The cause of this failure is that IFA is a syntactic technique: it is concerned only with the security classifications ('colours') of variables, not their values. This deficiency can be overcome by applying IFA to a high-level specification of the kernel (in which, for example, each regime is provided with its own set of general registers) rather than to the kernel implementation itself. The security of the implementation can then be established by showing it to be a ***correct*** implementation of the secure high-level specifications [23]. In conventional practice, however, this second stage is not performed. For KSOS, for example, only 'illustrative' proofs of the implementation were provided [7].

Because the KSOS kernel contains, among other things, a mechanism to support a multilevel secure file system, verification of the security of its high-level specifications is a significant task. It would be vastly more difficult and hugely expensive to verify the correctness of its implementation as well. Using a separation kernel, however, issues such as the verification of a multilevel file-server are factored out and handled separately from the verification of the kernel. Almost the entire activity of a separation kernel is concerned with the detailed management of features of the base hardware. In order to apply IFA, we must abstract away from these details and provide a high-level specification—whose verification would amount to little more than exhibiting a tautology. Almost the whole burden of verifying the security of the real kernel would then fall to the 'correctness' stage. While this procedure may be sound, it is very indirect and fails to provide one of the principal benefits we should desire of a kernel verification technique: a sharpened understanding of the issues that determine a kernel's 'security.'

A more conclusive argument against IFA as a verification technique for separation kernels is that it is incomplete: it does not address matters concerning the flow of control—in particular, the handling of interrupts. Recall that the SUE kernel does very little except field interrupts and allow one regime to SWAP control to another—and IFA provides no basis for the verification of these important and tricky matters. Questions relating to control flow cannot even be formulated within the mathematical model [9] that justifies IFA as a verification technique. In fact, it is doubtful whether that model really provides a sound basis for the verification of any sort of security kernel—but then it was not formulated for that purpose.

Feiertag's model was intended to provide a basis for verifying the 'Secure Object Manager' (SOM) of PSOS [22]—for which purpose it is eminently suitable. The model formulates a specification of multilevel security for a system which consumes inputs that are tagged with their security classifications and produces similarly tagged outputs. 'Ordinary' programs, such as the SOM or a file-server, are sound interpretations of this model. But a kernel is different. A kernel is essentially an abstract *interpreter*—it behaves like a hardware extension and executes instructions on behalf of its regimes. The identity of the regime on whose behalf it is operating at any time is not indicated by a tag affixed to the instruction by some external agent, but is determined by the kernel's own state.

To provide a sound basis for the verification of a kernel, we really need a model that captures its essential characteristics more completely and realistically. Robinson, one of those responsible for the verification of KSOS, has observed:

> "Despite current successes in proving that a given piece of kernel software provides security, it cannot be proven with existing techniques that there is no way to circumvent that piece of software. The answer may be to add some explicit notion of interpretation to the state machine model. This extended model would make it possible to address such concerns as parallelism, language semantics, and interrupt handling." [29]

A model with some of these characteristics is described in a companion paper to this [31] and is used to justify a new method for verifying kernels which enforce the policy of isolation. An informal explanation of this method is given in the next section.

Proof of Separability

The purpose of a separation kernel is to simulate a distributed environment. To the software in each regime, the environment provided by a separation kernel should be indistinguishable from that of an isolated machine dedicated to its private use. We can call this imaginary, private machine the 'abstract' machine for that regime, while the single, shared system that is actually available is called the 'concrete' machine. What we desire, for security, is that each regime's view of the concrete machine should exactly coincide with its own abstract machine. A similar requirement expresses the 'correctness' criterion for implementations of abstract data types. This latter criterion may be formulated precisely in terms of an 'abstraction function' [14]: that is, a function which maps from concrete to abstract states. The interesting feature of a separation kernel is that it is required to support *several* different abstractions simultaneously (a separate one for each regime) and it seems natural, therefore, to formulate the properties required of it in terms of *multiple* abstraction functions.

Take the simple case of a system supporting just two regimes—RED and BLACK. The abstraction function REDABS will map the states of the concrete machine into those of RED's abstract machine, while BLACKABS does likewise for BLACK. Now suppose the concrete machine performs some operation, COP, on behalf of the RED regime. We must require that the effects of this operation, as perceived by the RED regime, are just as if some operation REDOP had been performed by the RED abstract machine. Thus, if execution of COP takes the concrete machine from an initial state X to a final state Y, we demand that REDABS(Y) is exactly the same state of the RED abstract machine as that which results from applying the abstract operation REDOP to the abstract state REDABS(X). In other words, we require the following diagram to commute:

$$\begin{tikzcd}
{} \arrow[r, "\text{REDOP}"] & {} \\
{} \arrow[u, "\text{REDABS}"] \arrow[r, "\text{COP}"'] & {} \arrow[u, "\text{REDABS}"']
\end{tikzcd}$$

This condition ensures that the regime which is currently ‘active’ on the concrete machine cannot distinguish its actual environment from that of its abstract machine. But it is also crucial that the execution of a concrete operation on behalf of the active regime should not affect the state of the machine perceived by currently ‘inactive’ regimes. For isolation between RED and BLACK, therefore, we require that the concrete state transition from X to Y caused by executing COP on behalf of RED should cause ***no*** corresponding change in the states of inactive regimes. That is, we require that BLACKABS(X) = BLACKABS(Y), or in diagrammatic form:

$$\begin{tikzcd}
& {} & \\
{} \arrow[ur, "\text{BLACKABS}"] \arrow[rr, "\text{COP}"'] & & {} \arrow[ul, "\text{BLACKABS}"']
\end{tikzcd}$$

Because I/O devices can directly observe and change aspects of the concrete machine’s internal state (by reading and writing its device registers, for example), and can also influence its instruction sequencing mechanism (by raising interrupts), the activity of these devices is relevant to security. Consequently, we must impose conditions on their behaviour. Expressed informally (and only from the RED regime's point of view), these conditions are:

a) If REDABS(X) = REDABS(Y) and activity by a RED I/O device changes the state of the concrete machine from X to X', and the same activity will also change it from Y to Y', then REDABS(X') = REDABS(Y') (i.e., state changes in the RED regime caused by RED I/O activity must depend only on the activity itself and the previous state of the RED regime).

b) If activity by a non-RED I/O device changes the state of the concrete machine from X to Y, then REDABS(X) = REDABS(Y) (i.e., non-RED I/O devices cannot change the state of the RED regime).

c) If REDABS(X) = REDABS(Y), then any outputs produced by RED I/O devices must be the same in both cases.

d) If REDABS(X) = REDABS(Y), then the next operation executed on behalf of the RED regime must also be the same in both cases.

Conditions a) and b) above are the analogues, for I/O devices, of the conditions imposed on CPU operations by the commutative diagrams given earlier. All six conditions (the four above and the two expressed in the commutative diagrams) constitute the basis for a kernel verification technique which I call 'Proof of Separability.' A more precise statement of the six conditions may be found in the Appendix to this paper. A formal derivation of the six conditions, which attempts to demonstrate that they are exactly the ***right*** conditions, is given in [31], while the relationship between this method and verification by IFA is examined in [30], which also contains a small example of the application of the method. Description of a more realistic series of example applications is currently in preparation.

'Proof of Separability' seems to be technically superior to other methods for security kernel verification since it is based on a more realistic model and can address all the important issues, including those relating to interrupts, quite naturally. Also, it corresponds to a straightforward intuition about what security 'is' and encourages the kernel designer to examine his system from the viewpoint of each individual regime in order to ensure that the results of every action invoked by a regime are capable of ***complete*** description in terms of the objects known to that regime (and are invisible to all other regimes).

Conclusion

I have proposed an approach to the design and verification of secure systems which I suggest is particularly appropriate to small special-purpose applications. I advocate that secure systems should be conceived as distributed systems in which security is achieved partly by the physical separation of the individual components and partly by the trusted functions performed by some of those components. The task of specifying and verifying the properties required of the trusted components in order to achieve overall security should be tackled at this level of abstraction and on the assumption that components are physically isolated from one another. The purpose of a security kernel is simply to allow such a 'distributed' system to actually run within a single processor: its role is to provide each component of the system with an environment which is indistinguishable from that which would be provided by a truly and physically distributed system. Policy enforcement is not the concern of a security kernel. There is some similarity between these proposals and Popek's notion of 'levels of kernels' [26,27] while the idea that the management of shared resources can be handled by separate virtual machines can be traced back to Anderson [3].

This approach achieves a separation of concerns by completely decoupling the verification of the components which perform trusted functions from the verification of the security kernel. This latter task may be accomplished by a new verification technique which I call 'proof of separability.'

Application of these techniques should assist the development of systems whose security is based on simpler mechanisms and whose verification is correspondingly simpler, more complete and more compelling than is the case at present.

A Appendix

This appendix gives a more precise statement of the six conditions for 'Proof of Separability.' The statement is expressed in terms of a particular formal model for computer systems. Space permits only a terse description of the model here; a more complete description, together with arguments for its suitability and justification for the particular choice of conditions defining Proof of Separability may be found in [31].

The model comprises a finite set S of ***states*** and a set OPS $\subseteq S \to S$ of ***operations*** on those states. The system interacts with its environment by consuming elements of a set I of ***inputs*** and producing elements of a set O of ***outputs***. At each time step, the system emits an output and changes state. The output emitted depends upon the system's state and this action is modelled by the function OUTPUT : $S \to O$.

State changes occur in two stages: the first is caused by the receipt of an input, and the second by the selection and execution of an operation. The effect of receiving an input is modelled by the function INPUT : $S \times I \to S$, while the operation selection mechanism is modelled by the function NEXTOP : $S \to$ OPS. Thus, if the current state of the system is s and the current value of the input available from the environment is i, the system will emit the output OUTPUT(s) and move to the state NEXTOP$(s\bar{)}(s\bar{)}$, where $s\bar{=}$INPUT(s, i) is the intermediate state resulting from consumption of the input i.

A ***shared*** system supports a number of ‘users’ who are identified with a set C of ‘colours.’ Exactly one user is ‘active’ at any time: he is the user upon whose behalf instructions are currently being executed. The identity of the active user depends upon the state of the system at the instant when an operation is selected for execution. It is determined by the function COLOUR : $S \to C$.

The inputs and outputs of a shared system are composed of individual components which are ‘private’ to each user. The projection function EXTRACT is used to pick out components of a particular colour. Thus, when $c \in C$, $i \in I$, and $o \in O$, EXTRACT(c, i) and EXTRACT(c, o) denote the c-coloured components of the input ***i*** and the output ***o*** respectively.

For a shared system to be ***secure***, the input/output behaviour perceived by each user must be completely consistent with that which could be provided by a non-shared system dedicated to his exclusive use. This is achieved if each user $c \in C$ can produce a set S^c of c-coloured ‘abstract states’ and a set $\mathrm{OPS}^c \subseteq S^c \to S^c$ of c-coloured ‘abstract operations,’ together with ‘abstraction functions’

$$\Phi^c : S \to S^c$$

and

$$\mathrm{ABOP}^c : \mathrm{OPS} \to \mathrm{OPS}^c$$

which satisfy, $\forall c \in C, \forall s, s' \in S, \forall op \in \mathrm{OPS}, \forall i, i' \in I$:

1) $\mathrm{COLOUR}(s) = c \supset \Phi^c(op(s)) = \mathrm{ABOP}^c(op)(\Phi^c(s))$,

2) $\mathrm{COLOUR}(s) \neq c \supset \Phi^c(op(s)) = \Phi^c(s)$,

3) $\Phi^c(s) = \Phi^c(s') \supset \Phi^c(\mathrm{INPUT}(s, i)) = \Phi^c(\mathrm{INPUT}(s', i))$,

4) $\mathrm{EXTRACT}(c, i) = \mathrm{EXTRACT}(c, i') \supset \Phi^c(\mathrm{INPUT}(s, i)) = \Phi^c(\mathrm{INPUT}(s, i'))$,

5) $\Phi^c(s) = \Phi^c(s')$
$\supset \mathrm{EXTRACT}(c, \mathrm{OUTPUT}(s)) = \mathrm{EXTRACT}(c, \mathrm{OUTPUT}(s'))$,

6) $\mathrm{COLOUR}(s) = \mathrm{COLOUR}(s') = c \wedge \Phi^c(s) = \Phi^c(s')$
$\supset \mathrm{NEXTOP}(s) = \mathrm{NEXTOP}(s')$.

These are the formal statements of the six conditions for Proof of Separability. Conditions 1) and 2) correspond to the two commutative diagrams in the text, while conditions 3) to 6) correspond to those labelled a) to d) in the text.

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] S. R. Ames Jr. and J. G. Keeton-Williams. Demonstrating security for trusted applications on a security kernel base. In *Proceedings of the Symposium on Security and Privacy*, pages 145–156, Oakland, CA, April 1980. IEEE Computer Society.

[3] J. P. Anderson. Systems architecture for security and protection. In C. R. Renninger, editor, *Approaches to Privacy and Security in Computer Systems*, pages 49–50. NBS Special Publication 404, GPO SD Catalog No. C13.10:404, Washington, D.C., 1974.

[4] K. Auerbach. Secure personal computing (technical correspondence). *Communications of the ACM*, 23(1):36–37, January 1980.

[5] D. H. Barnes. Computer security in the RSRE PPSN. In *Networks '80*, pages 605–620. Online Conferences, June 1980.

[6] D. E. Bell and L. J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, Bedford, MA, March 1976.

[7] T. A. Berson and G. L. Barksdale Jr. KSOS—development methodology for a secure operating system. In *National Computer Conference*, volume 48, pages 365–371. AFIPS Conference Proceedings, 1979.

[8] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

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

[10] Ford. KSOS verification plan. Technical Report WDL-TR-7809, Ford Aerospace and Communications Corporation, Palo Alto, CA, March 1978.

[11] D. L. Golber. The SDC communications kernel, August 1981. Presented at DoD Computer Security Industry Seminar.

[12] B. D. Gold et al. A security retrofit of VM/370. In *National Computer Conference*, volume 48, pages 335–344. AFIPS Conference Proceedings, 1979.

[13] A. Hathaway. LSI guard system specification (type A). Technical Report Draft, MITRE Corporation, Bedford, MA, July 1980.

[14] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[15] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[16] C. E. Landwehr. Assertions for verification of multilevel secure military message systems. *ACM Software Engineering Notes*, 5(3):46–47, July 1980.

[17] S. B. Lipner. A comment on the confinement problem. In *Fifth ACM Symposium on Operating System Principles*, pages 192–196. ACM, 1975.

[18] A. F. Martin and J. K. Parks. Intelligent X25 level 2 line units for packet-switching. In *Networks '80*, pages 371–384. Online Conferences, 1980.

[19] E. J. McCauley and P. J. Drongowski. KSOS—the design of a secure operating system. In *National Computer Conference*, volume 48, pages 345–353. AFIPS Conference Proceedings, 1979.

[20] J. K. Millen. Security kernel validation in practice. *Communications of the ACM*, 19(5):243–250, May 1976.

[21] J. K. Millen. Operating system security verification. Technical Report M79-223, MITRE Corporation, Bedford, MA, September 1979.

[22] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical report, SRI International, May 1980. Second Edition, Report CSL-116.

[23] P. G. Neumann et al. Software development and proofs of multi-level security. In *Proc. 2nd International Conference on Software Engineering*, pages 421–428, San Francisco, CA, 1976.

[24] M. A. Padlipsky, K. J. Biba, and R. B. Neely. KSOS—computer network applications. In *National Computer Conference*, volume 48, pages 373–381. AFIPS Conference Proceedings, 1979.

[25] G. J. Popek et al. UCLA secure UNIX. In *National Computer Conference*, volume 48, pages 355–364. AFIPS Conference Proceedings, 1979.

[26] G. J. Popek and C. S. Kline. A verifiable protection system. In *Proc. International Conference on Reliable Software*, pages 294–304, Los Angeles, CA, 1975.

[27] G. J. Popek and C. S. Kline. Issues in kernel design. In *National Computer Conference*, volume 47, pages 1079–1086. AFIPS Conference Proceedings, 1978.

[28] Gerald J. Popek and David R. Farber. A model for verification of data security in operating systems. *Communications of the ACM*, 21(9):737–749, September 1978.

[29] L. Robinson. Quoted by P. zave in report of a panel session from specifications of reliable software conference, July 1979.

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

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

[32] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, February 1980.

[33] J. P. L. Woodward. Applications for multilevel secure operating systems. In *National Computer Conference*, volume 48, pages 319–328. AFIPS Conference Proceedings, 1979.