Abstract:

This paper surveys techniques for verifying security kernels. It begins with a tutorial description of two established methods for performing this task: these are 'access control verification' and 'verification by information flow analysis'. Both of these techniques are shown to be inadequate, on their own, to the task of verifying the security of a toy kernel which enforces a policy of isolation. A new method of verification by 'proof of separability' is proposed which deals with this problem quite simply and naturally: the basic idea is to prove that, to each of its users, the behaviour of the shared system is indistinguishable from that of an idealised (and manifestly secure) system which supports that user alone.

The application of this idea is then extended to more complex security policies and systems and is shown to lead to a clearer understanding of the role of a security kernel and of its relationship to 'trusted processes' and other security critical software within the system.

It is argued that the advantages of the verification and structuring techniques introduced here are such that they should substantially replace those used in current practice.

Verification of Secure Systems

By

J.M. Rushby

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick

Number 166
August, 1981

Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

bibliographical details

RUSHBY, John Martin

Verification of Secure Systems
[By] J.M. Rushby.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1981.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 166.)

Added entries UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 166.

Suggested classmarks (primary classmark underlined)

Dewey (18th):	001.64404	658.47
U.D.C.	519.687	519.718

Suggested keywords
INFORMATION FLOW ANALYSIS
PROOF OF SEPARABILITY
SECURITY KERNEL
VERIFICATION

Abstract

This paper surveys techniques for verifying security kernels. It begins with a tutorial description of two established methods for performing this task: these are 'access control verification' and 'verification by information flow analysis'. Both of these techniques are shown to be inadequate, on their own, to the task of verifying the security of a toy kernel which enforces a policy of isolation. A new method of verification by 'proof of separability' is proposed which deals with this problem quite simply and naturally: the basic idea is to prove that, to each of its users, the behaviour of the shared system is indistinguishable from that of an idealised (and manifestly secure) system which supports that user alone.

The application of this idea is then extended to more complex security policies and systems and is shown to lead to a clearer understanding of the role of a security kernel and of its relationship to 'trusted processes' and other security critical software within the system.

It is argued that the advantages of the verification and structuring techniques introduced here are such that they should substantially replace those used in current practice.

About the author

Dr. Rushby was a Research Associate in the Computing Laboratory from 1979-1982. He is now a Computer Scientist with SRI International, Menlo Park, California.

Verification of Secure Systems

(Preprint of Technical Report No. 166)

J.M. Rushby

Computing Laboratory
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
England

Tel.: Newcastle (0632) 329233
Arpanet mail: NUMAC at SRI-KL

1. INTRODUCTION

In this paper I shall describe, illustrate, and discuss some methods for proving that certain types of computer system are `secure´.

In the limited sense in which the term will be used here, a **secure** system is one which enforces certain specified restrictions on the ways in which the information it contains may be accessed, or may `flow´ within the system. Secure systems are of special interest and significance, not only because concern for security is growing in many areas of application, but also because they are among the first examples of real-world systems to which the techniques of formal specification and verification are being applied in earnest. They therefore provide both a stimulus to the development of these techniques and a test-bed for their evaluation.

Depending on the application, the leakage of personal or confidential information from a supposedly secure system could cause individual distress, financial loss, or the erosion of commercial, political or military advantage. In certain situations it could even endanger the national security itself. Systems of guaranteed security are required for those applications where such unauthorised disclosure of information cannot be countenanced. (At present, these applications are predominately military ones.)

If the cost of information leakage is great to those adversely affected by it, so its rewards are correspondingly high to those individuals, organisations, or governments who may benefit from the other´s loss. It follows that the `enemy´, as this adversary is generally known, will consider it worthwhile to devote considerable resources towards bringing that loss about. Any system responsible for the management of confidential information should be assumed, therefore, to be threatened by determined and skilled attempts to penetrate its defences - both by direct external attack and by infiltration or subversion of its own components.

A secure system is required to withstand such attacks and to preserve the confidentiality of information consigned to its care under all circumstances. This is a strong requirement; a system can only be trusted to this extent if its security is attested by utterly compelling evidence. A formal mathematical proof that the system conforms to an appropriate and precise specification of `secure behaviour´ could provide a sound basis for the provision of such assurances. This survey is concerned with the problems of constructing suitable proofs for the class of systems based on the `security kernel´ concept. (A different approach, requiring the verification of a complete operating system, is exemplified by PSOS [DELA79, NEUM77].)

Previous treatments of this topic have mostly proposed or expounded techniques in the context of their application to some particular, real system. Consequently, their descriptions have tended to be at a very general level [BERS79, POPE78b, SCHA77] and worked examples have been based upon a component of the system rather than the system as a whole [FEIE77, MILL76]. Consequently, it has not been easy for the reader to form a complete understanding of all the issues involved. Accordingly, my strategy here will be to examine a toy system (adapted from one due to Millen [MILL79]) which is sufficiently simple that it permits a comprehensive discussion and examination of its security properties. It was only by working through some analyses of this toy system that I began to understand the problems of security kernel verification for myself; I hope the reader will find the exercise similarly enlightening.

When I first performed these exercises, I came to the disturbing conclusion that existing approaches to the design and verification of secure systems were neither complete, nor altogether well founded. Consequently, I developed a new technique which is, I believe, more satisfactory than previous methods. A secondary purpose of this paper, therefore, is to point out what I see as the inadequacies of previous methods and to introduce my new approach and argue in its favour. As a consequence, this paper contains elements of both a tutorial and a polemic. Although I have tried to present the tutorial material as fairly as possible, I have not attempted to suppress my own point of view.

In order to make this paper accessible to those who have not encountered these topics before, the next sub-section provides a brief introduction to the notion of `security´ in its application to computer systems. This introduction is largely extracted from [RUSH81c] where the same material is covered at greater length. Other papers which survey similar material are [DENN79, LIND76] and [SHAN77]. An overview of the remaining sections of the paper follows this presentation of background material.

1.1. Background to Computer Security

Security is concerned with controlling access to information and with enforcing restrictions on its movement. The particular set of rules and restrictions to be enforced by a given system constitute its **security policy**. In the `pen and paper´ world, the best known security policies are those based on the military **multilevel** scheme. Here, in its simplest form, each document is given a **classification** chosen from the five hierarchically ordered levels: UNCLASSIFIED, RESTRICTED, CONFIDENTIAL, SECRET and TOP SECRET. (These are the levels used in Britain;

Americans omit the RESTRICTED level.) Individuals are given a **clearance** chosen from the same five levels and the policy requires that an individual may have access to a document only if his clearance equals or exceeds the classification of the document concerned. Thus, CONFIDENTIAL clearance allows access to UNCLASSIFIED, RESTRICTED and CONFIDENTIAL documents, but not to those classified SECRET or TOP SECRET.

The full multilevel scheme is actually more complex than this; indeed, some of its details are themselves classified. The main refinement to the basic scheme is the use of **categories** or **compartments** in order to provide more selective control of access to information on a `need to know´ basis. Thus, a person may be allowed access to SECRET level documents in the NATO and CRYPTO categories, but not to those in the ESPIONAGE category. In this extended form, the combination of a classification or clearance together with a set of categories constitutes a **security class**. A person may access a document only if his clearance equals or exceeds the classification of the document and if his category set includes that of the document. This right-to-access relationship on security classes has the mathematical properties of a partial order. An important special case of the multilevel policy is that in which the security classes are mutually incomparable. This is the policy of **isolation** where absolutely no access to information is permitted across the different classes.

Beyond this rather limited interpretation of security as a restriction on the movement of information, there are a number of related issues which are sometimes included under more general interpretations of the term. Examples include: preventing unauthorised modification or generation of information (this property is known as `integrity´ [BIBA77]), ensuring that no user can cause the denial of service to others, and enforcing the requirement that all who use system resources should pay for them. Other related topics include authentication of personnel (making sure a person is who he says he is) and the use of cryptosystems to provide secure communications. These are interesting and important matters but too remote from my central purpose to be discussed here. A useful source of information on these topics is the book by Hsiao, Kerr and Madnick [HSIA79], which also contains an extensive annotated bibliography.

Before proceeding to examine computer security specifically, it is necessary to recognise that concern for security must extend to the total system of which the computer is just a part. Security is a `weakest link´ property: it is no use protecting information inside the computer system if it is vulnerable elsewhere. Thus the entire organisation in which the computer system is employed must be scrutinized and steps taken to safeguard the security of its operations.

Until recently, many commercial and industrial installations had not adequately perceived the degree of threat to which their operations were exposed and were vulnerable to quite elementary attacks; contrary to popular belief, most `computer crime´ has not required the arcane skills of an evil `computer genius´ but has attacked security weaknesses in the larger system [BECK80, PARK76]. A whole `computer security´ industry has now sprung up to exploit these weaknesses in a more socially acceptable manner: by selling the expertise and gadgetry necessary to remove them. Most of the techniques employed derive from military and governmental practice, where the level of perceived threat is

far greater and where concern for security has existed for decades, if not centuries. Sophisticated systems of procedural controls have been developed, together with methods for ensuring the physical security of an installation and for vetting the trustworthiness of its personnel. Physical security, besides the obvious use of locks and guards, may, depending on the scale of perceived threat, go so far as to include the screening of cables and VDU's (in order to prevent their stray radiation from being picked up and decoded by an electronic eavesdropper) while procedural controls may require the immediate destruction of line-printer ribbons that have been used to print highly sensitive information, and the printing of that information on distinctive (coloured) stationery.

Once the security of the rest of the operation is tightened up, so the computer itself becomes the weakest link in the chain. The increasing dependency of almost all organisations on their computer operations further raises their vulnerability to attack from this quarter and consequently their need to examine and counter the threats which it may harbour. These threats may be internal as well as external to the system: attacks may be mounted by those with legitimate access to the system, or even by those involved in its design and implementation.

During the 1960's and 70's, considerable experience of the inability of conventional systems to withstand concerted and skilled attack was gained by performing **penetration audits**. These entail giving a `tiger team' of experts access to the system, but without granting them any special privileges denied to ordinary users. Within a few days, the team was invariably able to deliver, at will, a listing of any file held by the system or even to seize total control of its operations. The systems of all major manufacturers, including the specially `hardened' versions intended for secure operations, are believed to have been penetrated in this way - and then repenetrated after the flaws shown up by earlier penetrations had been fixed.*

The evidence of successful penetration audits is so alarming that the military authorities are forced to use slow, inflexible and costly procedural mechanisms such as **periods processing** in order to adequately safeguard the security of their operations. Periods processing involves dedicating the system to one security classification at a time. While the machine is running at SECRET level, all users not cleared to this level are refused access to the system and all data classified at levels other than SECRET are physically removed from the system. Following a SECRET period, the system must be `flushed' before bringing it up again at, say, the CONFIDENTIAL level. Flushing involves not only the exchange of all SECRET disks and tapes for CONFIDENTIAL ones, but the

---

* For obvious reasons, the details of military penetration audits are not reported in the open literature. However, it is noted in the report of the Anderson Panel that "none of the known tiger team efforts has failed to date" [ANDE72, vol. 1, p4] and that a large system which had been successfully penetrated required 10-15 man-years of effort and over 250 changes to the operating system in order to repair the deficiencies revealed by the penetration - and the `repaired' system was then repenetrated in less than one man-week of effort [ANDE72, vol. 1, p30]. For descriptions of some `civilian' penetration exercises, see [ATTA76, HEBB80, LIND75, WILK81].

writing of zeroes to all fixed storage (a thousand times - just to be sure) and the installation of a completely fresh copy of the operating system. Not only are these procedures exceedingly costly (the throughput achieved under periods processing is typically less than 10% of that obtained from the same hardware in normal operation) but they so hamper the timely and effective analysis of information that they threaten the effectiveness of the organisation in which they are used.

The principle reasons for the demonstrable insecurity of conventional systems are, firstly, that their security controls are constructed in an ad-hoc fashion, without the benefit of any formal understanding of what security really is, and, secondly, that privilege to alter or bypass the security controls is dispersed throughout their operating systems.

In order to perform their tasks efficiently (or at all) most operating system components require privileges denied to ordinary user processes. With most current hardware, special privileges are very much an all or nothing affair: in supervisor state anything is possible - and that includes the ability to bypass the security controls. So to believe that the security controls are always invoked correctly requires that we trust the entire operating system. This is unrealistic because the sheer scale and unmastered complexity of an operating system, as well as the brute nastiness of much of its interaction with the input/output system (self-modifying channel programs, for example), all conspire to make the existence of accidental security flaws a certainty, before we even contemplate the awful possibility that `trap doors´ might have been deliberately planted in the system during its construction. Worse yet, it is unrealistic to believe that security controls of guaranteed effectiveness can be retro-fitted to existing systems; there will be too many compromises embedded in its fundamental design and too much complexity to allow the ramifications of basic changes to be understood. Security, like reliability, is not an add-on feature: it must be designed in from the start.

These and other problems with the security of computer operations within military environments were reported in 1972 by the influential Anderson Panel [ANDE72], which was set up by the Electronic Systems Division of the United States Air Force. The Anderson Panel noted a pressing need for truly secure computing facilities (the cost to the USAF incurred by the absence of such facilities was estimated at one hundred million dollars a year [ANDE72, vol. 1, p28]) and observed that "patching of known faults in the design or implementation of existing systems without any better technical foundation than is presently available, is futile for achieving multilevel security" [ANDE72, vol. 2, p38].

In part, the Anderson Panel itself proposed a solution to some of the difficulties it had identified. This was the concept of a **reference monitor**: a single isolated mechanism that would mediate all accesses to data in order to enforce the given security policy. A reference monitor is required to be:

1) correct - it must correctly enforce the chosen security policy. It should be sufficiently small and simple that it can be subject to analysis and tests whose completeness is assured.

2) complete - it must mediate all accesses between subjects and objects. It must not be possible to bypass it.

3) tamper-proof - it must be protected against unauthorised modification.

The Anderson panel further identified the idea of a **security kernel** as a means of realising a reference monitor on conventional hardware. A security kernel may be considered as a stripped down operating system that manages the protection facilities provided by the hardware and contains only those functions necessary to achieve the three requirements listed above. In order to ensure that the kernel can be neither bypassed nor modified (requirements 2 and 3) all functions requiring special privileges or supervisor-state operation must be performed inside the kernel and subject, along with the rest of the kernel code, to the exhaustive scrutiny that is intended to guarantee its correctness. The rest of the system, and that should include most of the operating system as well as all user code, is constrained to operate in the protected environment maintained by the kernel. It may therefore be completely untrusted - for its only means of progress is by executing unprivileged hardware operations (whose effects are constrained by the protection facilities of the hardware, which are themselves under the control of the kernel) or by requesting service (via SVC or `trap´ instructions) directly from the kernel.

Ideally, the kernel contains only the code necessary to achieve the three requirements listed earlier. In practice, however, certain other operating system functions usually need to be brought inside the kernel interface in order to achieve acceptable performance on a conventional hardware base. This conflicts with the desire to keep the kernel minimally small and uncomplicated and makes the design of security kernels a task requiring exceptional skill and discipline. Nonetheless, a properly designed security kernel can be orders of magnitude smaller than a full general purpose operating system - small enough for its construction to be closely monitored (to avoid the infiltration of `trap doors´) and for there to be a real chance that it will function correctly.

Of course, those who place their trust in a security kernel require more than a real chance of its working correctly - they require guarantees. One approach to the provision of these guarantees might be to demonstrate that the kernel can withstand concerted attempts to defeat its security controls. Such penetration audits, however, while successful at revealing the inadequacies of conventional systems, are less suited to the more positive role of demonstrating the presence, rather than the absence, of security. The problem here, as with any approach based on testing, is its logical incompleteness: even if the system successfully withstands a variety of attacks, we know only that it is secure against those attacks actually tried - we have no assurance of its invulnerability to other attacks. Consequently, attention has now turned to the possibility of proving (or **verifying**) the security of a kernel. By this is meant a formal, mathematical demonstration that the kernel enforces an appropriate and precise specification of `security´.

This is an attractive idea: if performed successfully it would surely provide the totally compelling evidence that justifies the trust placed in a secure system. (Furthermore, the discipline imposed on the

system design process by the necessity of proof is widely considered to be beneficial in its own right.)

In order to verify a security kernel, it is necessary to have a precise, formal specification of what it means for a system to be `secure´. (In fact, it is hard to see how any approach can progress without such a specification.) A conventional security policy does not constitute an adequate specification from this point of view. Rather, it has more the flavour of a high level `requirement´ since it usually deals in concepts from the `pen and paper´ world (such as `access´ to a document) which have a less clear-cut meaning in the context of a computer system. The role of a security specification is, in part, to elaborate what it means to `access´ information and for information to `flow´. As will become apparent later, these are not such simple notions as they may at first seem.

The earliest attempts to specify system properties relevant to security dealt with `protection´ rather than `security´ itself; that is to say, they were concerned with mechanisms to limit and control access to the physical resources of the system, rather than with the more abstract problem of controlling the flow of information. The idea of an `access matrix´, for example, was introduced by Lampson in 1971 [LAMP71] in order to model the most basic of security, or protection, requirements: the right of each user to control access by others to his private files. The access matrix is a two-dimensional array indexed, in its simplest manifestation, by users (rows) and files (columns). The entry in the i´th row and j´th column records the type of access allowed by user i to file j. The types of access that may be indicated in the matrix cells include the familiar ones: read, write, append and probably some additional ones indicating the right to change the matrix entries themselves, or to give some rights away to other users. In its more elaborate forms, the columns and rows of the access matrix may be indexed by processes as well as by files and users. Matrix entries may then indicate the right of one process to call another, or to establish communication with it. In this form, the entities that index the rows of the matrix are called subjects (active entities) while those that index the columns are called objects (passive entities). Certain entities (such as processes) may partake of both attributes.

Associated with the access matrix is the policy of `data security´ which requires that no access between a subject and an object may take place unless the appropriate cell of the access matrix explicitly records that the type of access concerned is to be allowed.

The access matrix model of data security cannot be implemented directly in any practical system since the underlying hardware will offer a very different, and usually very crude, set of protection facilities. Thus the access matrix constitutes an ideal reference (a specification) against which to judge a practical implementation that must achieve equivalent behaviour by manipulation of capabilities, storage keys, relocation registers or whatever is provided by the hardware.

The access matrix model of data security is a purely **discretionary** one. That is, each user may, at his own discretion, grant or deny permission for others to access his files. To see why this is an inadequate formulation of security from the military point of view (among others), consider a user cleared to TOP SECRET level who owns a file

containing TOP SECRET information. Under the simple data security policy he could, at his discretion, give permission for an UNCLASSIFIED user to read that file - and this would obviously constitute a breach of multilevel security, if not of data security. Of course it can be argued that a user would not do such a thing if he were cleared to TOP SECRET level - it is the purpose of clearances to identify those who may be trusted. And, in any case, such a security breach is not specifically a problem of computer security: a person who will give access to a computer file might also hand over physical files too. The only threat we need to counter, so this argument might go, is from the UNCLASSIFIED user trying to gain access to the TOP SECRET file - and data security will provide this level of security since we can trust the file's owner not to give UNCLASSIFIED users the necessary access permission.

This argument is sound, so far as it goes, but it misses one vital point - a point that is really the whole basis of the computer security problem. Suppose our TOP SECRET user were editing his file. Then he would be using the system's editor program which, because it would be acting on his behalf, would partake of the same access rights as himself. (The editor must obviously be able to read and write the file it is manipulating). But the editor is not a trusted user with TOP SECRET clearance - it is a utility program, probably supplied by an outside vendor and subject to casual modification by systems programmers. It may easily have been modified, or even written, by the enemy and could, while it has legitimate access to the TOP SECRET file, attempt to make it accessible to an UNCLASSIFIED user.

This possibility that the computer system itself might contain software with unexpected side effects (this is the `Trojan Horse' threat) leads to the conclusion that access control cannot be purely discretionary: it must be reinforced by additional mechanisms that are cognizant of the (nondiscretionary) multilevel security policy. We must be sure that a TOP SECRET file can <u>never</u> be read by an UNCLASSIFIED user.

One of the earliest, and certainly the best known, attempt to specify the nondiscretionary aspect of multilevel security in its application to computer systems was performed by Bell and La Padula at the Mitre Corporation in the early 1970's. The work is described in a number of reports, of which [BELL76] is the most complete and self-contained.

Bell and La Padula began by taking the access matrix model and adding to it a record of the classification of each object and of the <u>current</u> and the <u>maximum</u> clearance of each subject. (A TOP SECRET user may not always wish to sign on at that exalted level and so it is useful to distinguish the level at which he is currently operating from his maximum permitted level.) The property to be enforced, in addition to discretionary data security, is that no user may gain read access to any object that exceeds his current clearance. This is called the **simple security property** (or **ss-property**) and Bell and La Padula report [BELL76, p16] that in their early efforts they considered it to capture the whole of multilevel security.

Further consideration of the Trojan Horse threat described earlier will show that this belief is misplaced, however. If the editor program, while it has legitimate access to the TOP SECRET file, places a

copy of it in an UNCLASSIFIED location, where it can be read by UNCLASSIFIED users, then security has been breached just as surely as if those UNCLASSIFIED users gained access to the TOP SECRET file directly - even though the ss-property has not been violated.

It was to cope with this problem that Bell and La Padula formulated their famous ***-property** (pronounced `star-property´) which states that if a subject has, simultaneously, read-access to an object X and write-access to an object Y, then the classification level of Y must be greater than or equal to that of X.

Bell and La Padula´s model of multilevel security rapidly became well known and widely accepted: those who know nothing else of security have generally heard of the *-property. In addition to the ss- and *-properties, their model contained other properties, dealing with the creation and deletion of objects, and also imposed a hierarchical structure on the objects. These aspects of their model are rather specific to the Multics system which was its intended application.

If we accept Bell and La Padula´s model as an adequate formulation of multilevel security, then it would seem that a kernel can be proved secure by showing that it correctly enforces the requirements of the model on all accesses by users to data. Proving that all accesses to data are in accord with some precisely stated policy is called **access control verification.** Unfortunately, it was discovered during one of its earliest applications [SCHI75] that this method of verification is insufficient to guarantee the absence of undesired information flow: a kernel, even one whose access controls have been verified, can itself be used as a path for insecure information flow! (I will illustrate how this can occur in Section 3.)

Lipner [LIPN75] and Millen [MILL76] were the first to argue that these `leakage paths´ result from a failure to identify all the objects within the system in sufficient detail. Objects - the repositories of information - must be considered to include, not just files, but all storage locations within the system, including the kernel´s own variables. The Bell and La Padula model of security remained an adequate one, they argued; it was merely necessary to refine its `granularity´ and assign appropriate security classifications to all kernel variables and to ensure that the kernel obeyed the ss- and *-properties in its own accesses to these variables. Obviously, the kernel cannot monitor its own accesses at run-time and so the necessary guarantees must be obtained by compile-time analysis of the kernel code. This analysis can be accomplished by a method of **information flow analysis** which was first proposed by D.E. Denning [DENN75, DENN76]. A theoretical justification for this approach was provided by Feiertag and his co-workers [FEIE77] who gave a highly abstract specification of what is meant by multilevel security and then went on to deduce that properties equivalent to the ss- and *-properties, and their verification by information flow analysis, are sufficient to guarantee this notion of security.

The techniques of access control verification and information flow analysis have been used, in one form or another, in several recent systems. These include UCLA Secure UNIX [POPE78b, POPE79, WALK80], KVM/370 [GOLD77, GOLD79, SCHA77], and KSOS [BERS79, McCA79a]. The first part of the rest of this paper is largely concerned with an exposition and examination of these techniques.

1.2. Overview of the Remaining Sections

In the following sections of this paper I shall describe and illustrate the techniques of access control verification and information flow analysis; I will then proceed to discuss the problems that still remain in the area of secure system verification and will propose new techniques for their resolution.

I shall illustrate the various techniques by applying them to a very simple kernel design which will be described in the following section. In order to keep the illustrations uncomplicated, the early sections of this paper will consider only the most basic of all security policies: that of isolation. This policy requires the total absence of information flow between different users of the system and may be considered as a degenerate case of the multilevel security policy. This simple policy permits a straightforward presentation of the ideas underlying access control verification and information flow analysis yet still allows some of their weaknesses to be exposed quite clearly.

The weakness of access control verification, which will be discussed in Section 3, is simply that it does not address the issue of information flow - it is only concerned with access to the repositories of information. It cannot, therefore, prevent the `leakage' of information. The absence of leakage channels can be established using the technique of information flow analysis - and this is the subject of Section 4. However, this method is inherently syntactic (that is, it considers only the security classifications of variables, not their actual values) and cannot cope with implementation-level descriptions of a kernel, only with its high-level specifications. The security of an actual implementation must be established by a two-stage process: first information flow analysis is used to verify the security of a `design specification' and then the implementation is proved to be correct with respect to that specification. Techniques for proving the correctness of an implementation with respect to its specifications are fairly well understood, if little practised.

I will argue, however, that the contribution made by information flow analysis to this two-stage demonstration of security is small: the brunt of the argument is borne by the correctness verification stage. This is unfortunate, since only the information flow analysis step is actually performed in conventional practice. Accordingly, I will present, in Section 5, a new method of security kernel verification which is sufficient, on its own, to verify the security of those kernels which enforce the policy of isolation. The new method, which I call **Proof of Separability** and which is given a formal description and justification in [RUSH81b], is really no more than a restricted and specialised application of conventional correctness verification. Its basis is to prove that to each of its users, the behaviour of the actual system is indistinguishable from that which would be provided by an idealized (and manifestly secure) system which is dedicated to that user alone.

In Section 6, I shall return to the examination of information flow analysis and, in particular, to the `correctness' verification that must be performed if the security of a kernel implementation is to be inferred from the security of its specification. I will illustrate the claim made earlier that, at least in the case of kernels which enforce

the policy of isolation, the contribution made by information flow analysis to the overall demonstration of security is negligible. I will then show that the ordinary `correctness´ verification - which contributes the greater part of the complete proof of security in this case - is most conveniently organised in such a way that is becomes very similar to proof of separability. There is a significant difference between the two techniques, however, in that proof of separability is able to address important matters relating to I/O devices, interrupt handling, and the flow of control, that are beyond the scope of the other technique.

Upto this point, only the security policy of isolation will have been considered. The additional problems that arise in the design and verification of systems intended to enforce more complex policies will be considered in Section 7. The first of the more complex types of policy to be examined will be one which arises in connection with the design of `secure front ends´ for network applications [AUER80, BARN80]. These systems are composed, at their simplest, of two components identified as `red´ and `black´ which are allowed to exchange information with each other, but only over a single known channel (whose bandwidth is limited and whose traffic may be monitored). The concern here is not whether red and black can communicate, but rather what channels are available for their communication. Consequently, I shall call this type of security policy one of **channel control** since the goal of verification in this instance is to prove the absence of any communications channels other than those intended.

I will demonstrate how a channel-control policy can be verified by the tactic of first `cutting´ the allowed communications channels and then verifying the system that results from this surgery with respect to the policy of isolation: only if there were no illegitimate channels in the original system will cutting the legitimate channels cause its components to become totally isolated from one another.

From channel-control policies I will progress, in the middle part of Section 7, to a consideration of multilevel policies. This part of the paper will be more controversial than earlier sections because I consider established approaches to the design and verification of multilevel secure systems (and, indeed, much of the current conception of the nature and role of a security kernel) to be inappropriate.

Current approaches to the design of multilevel secure systems, exemplified by KSOS [McCA79a, McCA79b], conceive of the security kernel as a centralized agent for the enforcement of a single system-wide (multilevel) security policy. This approach inevitably leads to difficulties since practical systems necessarily provide a number of functions that cannot be accommodated within that discipline. Examples from KSOS include "secure spoolers for line-printer output, dump/restore programs, portions of the interface to a packet-switched communications network etc." [BERS79, page 365]. In order to provide these essential functions, it has been found necessary to introduce `trusted processes´ into KSOS and other kernelized systems. Trusted processes are not part of the kernel but are accorded special privileges to evade or override the security controls normally enforced by the kernel.

Once trusted processes are admitted to the system, the kernel is no longer the sole determinant of overall 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 have to 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!

I shall argue that the roots of these difficulties with trusted processes lie in the mistaken belief that a single security policy can be applied uniformly to all the components of a secure system: the detailed security requirements of a file handler and a line-printer spooler are inescapably different - even when both form part of a system intended to enforce one multilevel security policy. The imposition of the *-property, say, throughout the whole system overconstrains the behaviour of certain components, so that they require special privileges in order to perform their allotted tasks, while at the same time failing to provide the controls that really are required. One of the important properties of a secure line-printer spooler, for example, is that it should correctly identify the security class of each item of output which it produces.

I will propose that the different security requirements of individual system components should be recognized and responsibility for their enforcement devolved to the components themselves: the system should be conceived as a network of distributed, independent, yet co-operating components, each obeying a security policy appropriate to its own function within the larger scheme. 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.

Were such a system to be realised as a physically distributed one, there would be no need for a security kernel: security would be achieved partly by the physical separation of its individual components and partly by security critical functions performed by some of those components. A security kernel is only needed when it is decided to support all the components of the system within a single, shared processor. In this case, the role of the security kernel 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 such a security kernel.

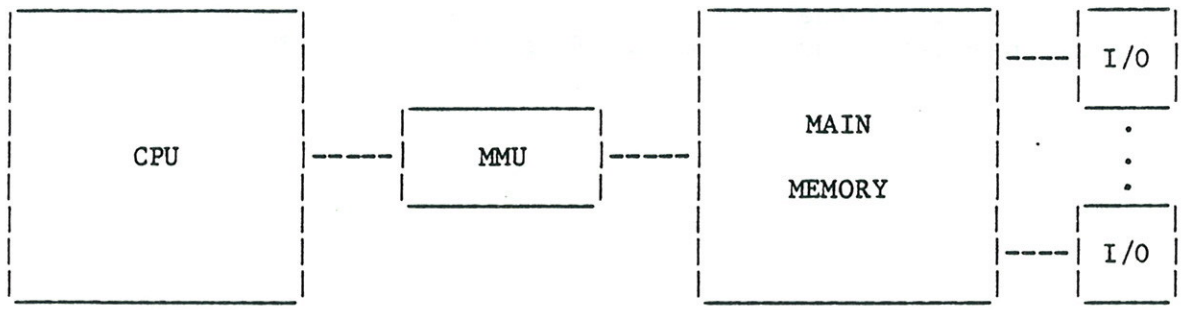
This approach leads to a clean separation of concerns: those issues which are due to the policy and function of the system are completely decoupled from those which are simply consequences of the fact that conceptually distinct components of the system all share a single processor. Specification and verification of the critical components of the system may be accomplished by a variety of techniques: information flow analysis may be appropriate for some types of component, while others are best served by techniques developed for `ordinary´ functional correctness. The kernel may be verified by proof of separability.

The third and final part of Section 7 will briefly sketch the application of these ideas to the specialized security policies found in systems such as network `guards´ and `filters´. Lastly, in Section 8, I will present some conclusions drawn from the preceding exercises and arguments.

In order to make them accessible to a wide audience, the descriptions and examples contained in this paper will be informal and will involve only the amount of detail needed to expose the main issues. A simple `toy´ kernel will serve as a concrete basis for the discussion and a description of this kernel forms the next section.

2. AN EXAMPLE OF A KERNEL DESIGN

The toy kernel design that I shall use is adapted from [MILL79] (as also are Section 3 and the first part of Section 4). The kernel is to run on a hardware configuration consisting of a CPU connected to a main memory via a memory management unit (MMU). I/O devices are connected directly to the main memory. (That is, I/O device registers are located at fixed addresses in memory - PDP-11 style.)



The CPU contains eight **general registers** named R(0),R(1),...,R(7), while the MMU contains four **mapping registers** denoted MAP(0),...,MAP(3). The machine has two modes of operation: privileged and unprivileged; the contents of the mapping registers cannot be changed in unprivileged mode. The main memory consists of 128 **blocks**, each of 1024 **words**. Word w of block b is denoted MEM(b,w). A block of memory is accessible to the CPU only if its block number is loaded into one of the mapping registers. (This is the basis of the protection mechanism provided by the hardware.)

The system which runs on this machine is intended to provide a service to a number of different users while enforcing a security policy of isolation: it must be impossible for different users to communicate with each other via the system. This policy has been chosen for study because it is the simplest possible and also because it is a special case of both the multilevel and channel control types of security policy. Techniques which work for this case may prove capable of extension to either of the two more general types of policy; conversely, techniques which prove inadequate in this simple case must surely be viewed with suspicion.

Users communicate with the system through its I/O devices - each I/O device is dedicated to one particular user and all the devices attached to addresses in any one memory block are dedicated to the same user. The environment or `virtual machine´ perceived by each user will be said to constitute his **regime**.

The interface presented by the system to the user regimes consists of the operations of the unprivileged instruction set together with those provided by the kernel (via `trap´ or `SVC´ instructions). In order to examine the security provided by this interface we must have some precise description of the effects of its operations. The most accurate descriptions are presumably those closest to the actual implementations of the operations - perhaps assembler code listings for the kernel operations and logic diagrams for those provided by the hardware - but it is not easy to reason formally with such low-level descriptions. Instead, it seems preferable to reason about the security of the

system with respect to some higher-level, more abstract, description of its behaviour and then to handle the problem of ensuring consistency between this descriptive level and the actual implementation as a separate issue (which will not be addressed here).

The descriptive method used in this paper is based on a non-procedural specification technique due to Parnas [PARN72] and is superficially similar to the specification language SPECIAL designed at SRI [ROUB77]. I have used this specification technique simply to maintain consistency with most of the literature on kernel verification. Several other specification techniques and languages have been proposed (see, for example [ABRI80, BURS81, LOCA80, MUSS80a, NAKA80]), and some of these may prove superior to the current method - but that is a topic which requires separate study.

The behaviour of our system will be described in terms of a set of **state variables** and the effect of each operation will be specified by a set of equations which define the values of the state variables after execution of the operation in terms of their values before execution. Quoted names (e.g. X´) are used to indicate the new (post execution) values of state variables while unquoted names (e.g. X) denote the old (prior to execution) values. Thus the equation

$$X' = X + Z$$

specifies that the new value of the variable X is to be equal to the sum of its prior value and that of the variable Z. Sets of equations involving subscripted variables may be abbreviated as follows, for example:

$$(\forall i)\ R(i)' = 0.$$

The range of the bound variable i will usually be defined elsewhere. Here it may be assumed to be 0...7 since these are the indices of the general registers. (Recall these are denoted R(0),...,R(7).)

The equations defining an operation are gathered together in the **EFFECTS** section of its specification and are to be interpreted as mathematical equations; they are not programming `statements´ to be executed in some order. If the **EFFECTS** section of a specification does not define a new value for a particular state variable, then the value of that variable is unchanged by the operation.

Preceding the **EFFECTS** section of a specification is the (optional) **PRECONDITIONS** section which consists of a number of predicates over the state variables defining the conditions which must be satisfied if execution of the operation is to proceed normally. These predicates are evaluated one at a time, in the order in which they appear, and should any of them yield the value `false´, then execution of the operation terminates immediately with the values of all state variables unchanged.

We are now in a position to define the unprivileged hardware operations of our system.

The essence of the hardware protection offered by our machine is that data can only be transferred between main memory and the CPU via the MMU. This is reflected in the specifications of the following two operations which are the only operations that reference the main memory.

```
OPERATION LOAD(i,j,w)
EFFECTS
    R(i)' = MEM(MAP(j),w)
END

OPERATION STORE(i,j,w)
EFFECTS
    MEM(MAP(j),w)' = R(i)
END
```

Informally, LOAD(i,j,w) loads general register i with word w of the memory block whose name is loaded in the j'th mapping register, while STORE(i,j,w) transfers data in the opposite direction.

Presumably our machine also has operations for moving data among the general registers and for performing computations and tests upon their contents. The precise details of these operations are unnecessary to our purpose and need not be specified here - but they would certainly be required if verification were to be performed in earnest.

Now let us consider the operations provided by the kernel itself. The kernel is intended to support a fixed number (n) of user regimes which are identified by the integers 0,1,...,n-1. A policy of isolation is to be enforced and so each regime is associated with its own unique security class. For greater vividness, I shall usually refer to the security class associated with a regime as its `colour'. (This terminology, and also the choice of `red' and `black' as representative colours, stems from cryptological usages.) Since, under a policy of isolation, a regime's identity uniquely determines its colour, there is no real need to distinguish between these two attributes. However, it seems more convenient to do so and consequently the state variable RCOLOUR is introduced: RCOLOUR(r) records the colour of regime number r.

At any instant, exactly one regime is `active' - it is actually running on the CPU. The number of this regime is recorded in the variable AR. The active regime may relinquish the CPU to another by executing a SWAP operation. When a regime SWAP's out, the contents of its general and mapping registers are saved in order that they may be restored when the regime next becomes active. The variables SR and SMAP are used as save areas for the general and mapping registers, respectively.

Memory blocks are either `free' or are owned by a particular regime. The variable BCOLOUR(b) contains the colour of the regime which owns block b, or the pseudo-colour `SYSTEM' if this block is free. Since the number of mapping registers is limited, a regime may be unable to access all of its memory blocks simultaneously: it can only access those whose block numbers are loaded into one of the mapping registers. This information is recorded in the Boolean variable ACCESSIBLE(b) - which has the value true if and only if block b is accessible to its owner. A regime may request one of its currently inaccessible blocks to be made accessible by performing an ATTACH operation: ATTACH(b,j) asks

for memory block b to be made accessible through mapping register j - the block previously attached to that register then becomes inaccessible.

A request to gain ownership of a free block may be made by means of the ACQUIRE operation, while a block which is owned but inaccessible may be returned to the free pool by means of the RELEASE operation. Not all blocks may be RELEASEd by their owners, however. A block to which an I/O device is attached may not be released - for if it were, and some other regime were subsequently to ACQUIRE it, then that regime would also gain control of the attached I/O device. This cannot be allowed and so the Boolean state variable FIXED(b) is introduced to identify those blocks which may not be RELEASEd (indicated by FIXED(b) having the value `true´).

The formal specifications of the four kernel operations are given in Figure 1, which also contains a list of the system state variables. To avoid repeated quantification of the ranges of index variables (i.e. subscripts), the symbols r,b,w,i and j will be used consistently to stand for variables in the following fixed ranges:

INDEX	RANGE	USAGE
r	0 ... n-1	regime number
b	0 ... 127	block number
w	0 ... 1023	displacement of word within block
i	0 ... 7	general register number
j	0 ... 3	mapping register number

The specifications given in Figure 1 are incomplete in one important particular: no constraints are placed upon the initial system state. It is clear that some constraints are necessary - we must require, for example, that each memory block is accessible to at most one regime. For tutorial purposes, however, it is unnecessary to spell these initial conditions out in detail - although they would be an important feature of any complete proof.

Also absent from the description of the system is any mention of program control: there is no indication of how operations are selected for execution. Presumably this is performed under control of a program counter (and, possibly, an interrupt system) but to include a precise explanation of how this is done would introduce considerable, and at this stage unnecessary, complexity into our system description.

It seems necessary to end this section with a word of warning to the reader: although the system specified in Figure 1 <u>is</u> insecure, its insecurity is not intended to be either trivial or obvious. Any reader who is previously unacquainted with these topics and who believes he can see the insecurity directly, is urged to re-examine his understanding of the system´s behaviour.

State Variables

MEM(b,w)	word w of block b of main memory
R(i)	i´th general register
MAP(j)	j´th mapping register
SR(r,i)	copy of R(i) saved on behalf of regime r
SMAP(r,j)	copy of MAP(j) saved on behalf of regime r
RCOLOUR(r)	colour of regime r
BCOLOUR(b)	colour of block b (=SYSTEM if block b is free)
ACCESSIBLE(b)	**true** if block b is accessible to its owner
FIXED(b)	**true** if block b may not be released
AR	identity of the active regime

Operations

```
OPERATION SWAP    {relinquish CPU}
EFFECTS
    AR´ = AR + 1 (mod n)
    (∀i) R(i)´ = SR(AR´,i)
    (∀i) SR(AR,i)´ = R(i)
    (∀j) MAP(j)´ = SMAP(AR´,j)
    (∀j) SMAP(AR,j)´ = MAP(j)
END

OPERATION ATTACH(b,j)   {attach block b to mapping register j}
PRECONDITIONS
    BCOLOUR(b) = RCOLOUR(AR)
    not ACCESSIBLE(b)
EFFECTS
    MAP(j)´ = b
    ACCESSIBLE(MAP(j))´ = false
    ACCESSIBLE(b)´ = true
END

OPERATION ACQUIRE(b)   {request ownership of block b}
PRECONDITIONS
    BCOLOUR(b) = SYSTEM
EFFECTS
    BCOLOUR(b)´ = RCOLOUR(AR)
    (∀w) MEM(b,w)´ = 0
END

OPERATION RELEASE(b)   {relinquish ownership of block b}
PRECONDITIONS
    BCOLOUR(b) = RCOLOUR(AR)
    not ACCESSIBLE(b)
    not FIXED(b)
EFFECTS
    BCOLOUR(b)´ = SYSTEM
END
```

Figure 1 : Kernel Specification of Example System

3. ACCESS CONTROL VERIFICATION

We would like to prove that our toy kernel enforces total isolation between its regimes. Isolation means the complete absence of any information flow from one regime to another and so it seems that we need to prove something about the way our kernel controls the flow of information. However, this seems a rather elusive property and so it might be best to begin, not by tackling the problem of information flow directly, but the problem of controlling access to the places where information is stored. In our toy system, for example, it is certainly necessary to control access to memory blocks - for it is clear that security is compromised immediately one regime accesses the private memory of another. One property we must require of the system, therefore, is that regimes may only gain access to memory blocks of their own colour. A requirement such as this is called an `access control policy' - in contrast to the information flow type of security policy we have implicitly considered until now. Access control policies are not concerned with information or information flow directly, but with control of access to the repositories of information.

In order to prove that the toy system satisfies the policy of only allowing regimes to access memory blocks of their own colour, we must first state that policy more formally and precisely. Careful examination of the hardware protection facilities provided by the system reveals that a regime can access a particular memory block only if:

a) it is the active regime, and

b) the name of the memory block is loaded into one of the mapping registers.

Thus, the access control policy may be formally stated as:

$$(\forall j)\ \mathrm{BCOLOUR}(\mathrm{MAP}(j)) = \mathrm{RCOLOUR}(\mathrm{AR}). \qquad (*)$$

That is, the colour of any accessible block must be the same as the colour of the active regime.

To verify our system with respect to this policy, we must prove that (*) is an **invariant** of the system: that is to say it is a property which is preserved by each and every operation of the system.

The technique for proving the invariance of (*) is a conceptually straightforward induction: we prove that (*) is true of the initial system state and then, for each operation specified in the system, we prove that if (*) is true before execution of the operation, then it will also be true after its execution. The details of the proof, however, are quite complex and require a fair degree of inventiveness to carry out. This, as is so often the case with inductive proofs, is due to the fact that it is necessary to prove a rather stronger theorem than that actually required. Thus, in order to prove the invariance of (*), we also need to prove the invariance of several other assertions which state, for example, that the saved versions of the mapping registers also obey a property similar to (*) and that each block number appears in at most one mapping register, whether saved or actual. Because they are inter-

dependent, the invariance of all these assertions must be established simultaneously by a single induction. I shall omit the details. (A similar example is partially worked through in some detail by Millen [MILL79].)

After performing access control verification for the toy system, we may be sure that no gross security compromises can occur - no regime can directly handle the private memory of another. However, the access control policy we have chosen in (*) is a rather weak property and information may still be able to leak between regimes by indirect means. Consider, for example, the general CPU registers of the system. They do not feature in the statements of any of the invariants proved during access control verification and, in fact, the proof of the access control policy (*) remains valid when the two equations

$$(\forall i)\ R(i)' = SR(AR',i)$$
$$(\forall i)\ SR(AR,i)' = R(i)$$

(dealing with the saving and restoration of the general registers) are deleted from the specification of the SWAP operation. But in the absence of these two equations, the system is manifestly insecure - the contents of the general registers persist through a regime swap unchanged and so allow the active regime to pass information to its successor.

Another example of the inadequacy of our access control policy concerns the zeroing of memory blocks that takes place during an ACQUIRE operation. This action is also irrelevant to the access control policy and may be deleted without affecting its proof - yet it is clearly vital to security since, in its absence, information left recorded in a RELEASEd block is available to the next regime to ACQUIRE it.

Plainly, there is more to the security of our toy system than the invariance of the access control policy given in (*).

The question remains whether this inadequacy is due to the particular policy statement embodied in (*) or to a weakness inherent in the method of access control verification itself. Certainly the policy statement given by the invariant (*) is rather limited and weak and it is clear that some additional restrictions should be imposed in order to exclude security flaws such as those described above. Some versions of the Bell and La Padula model, for example, include a requirement which Feiertag et al. [FEIE77] state as follows :

> "REWRITING OF NEWLY ACTIVATED OBJECTS -- A newly activated object is given an initial state that is independent of all the state of any previous incarnation of the object."

If interpreted appropriately, this rule prohibits the second security flaw described above (failure to zero the contents of a newly ACQUIREd block), but it still fails to catch the first one (omitting to save and restore the contents of the general registers on a regime SWAP). Of course, special ad-hoc rules can be constructed for this and other cases too, but there seems to be no systematic way of enumerating rules to cover all the different `leakage channels´ through which information might flow from one regime to another.

Leakage channels need not be so obvious as those just described. In fact, they can be quite subtle and present just the type of insidious threat that security verification is intended to exclude. As an example, consider a multilevel secure system that allows the discretionary sharing of files (subject to the constraints of multilevel security, of course). Imagine a user A with legitimate access to a TOP SECRET file X and suppose that A wishes to leak the contents of this file to an UNCLASSIFIED user B who is running concurrently. Suppose, further, that A owns an UNCLASSIFIED file U which B repeatedly tries to access. These accesses will succeed only if A has granted permission for B to read the file U (which he may legitimately do). Of course it would violate the *-property if A copied the contents of X directly into U, but there is nothing to stop him from repeatedly granting or denying permission for B to access U. Since B can determine whether his accesses to U succeed or not, he has the ability to sense this activity by A. A can therefore `tap out´ messages to B through the pattern by which he grants and denies permission for B to access U. A communication channel, albeit a rather slow and noisy one, has been established from A to B and it is a simple matter for A to transmit the contents of X over this channel. Notice that no violations of the ss- or *-properties have occurred; the kernel has not failed, it has merely been duped by a technique it was not designed to counter.

Leakage channels of this type might seem merely curious and hardly a serious threat, but this would be a mistaken view. Channels capable of driving user terminals through just this kind of channel have been established in operational systems (two examples from Multics are cited by Popek and Kline [POPE78a]) and bandwidths far lower than this are unacceptable in military systems.

Accordingly, we must conclude that access control, while it is plainly a necessary feature of any system that could be called `secure´, is not a sufficient mechanism for the exclusion of undesired information flows. Access control verification, therefore, cannot be used on its own to certify the security of systems which are required to prevent such flows. One possibility, however, is that access control verification could be used to guarantee the absence of the more gross and direct forms of security flaw, while other (probably informal) techniques are used to check the absence of leakage paths.

Although it is hard to discover from the open literature precisely what verification technique has been used with KVM/370 (a kernelized secure version of VM/370 developed by the System Development Corporation [GOLD77, GOLD79, SCHA77]) the impression given (most notably by Schaefer et al. [SCHA77]) is that it is exactly this combination of access control verification coupled with informal checks for leakage channels. Of course, this technique does not provide the totally compelling evidence that we should expect of an ideal security verification technique, but it is surely better than one based wholly on informal procedures.

If access control verification is inadequate to guarantee a security policy that is concerned with information flow, then we must surely seek either a different verification technique, or else a different type of security policy. The rest of this paper is concerned with techniques which are (claimed to be) adequate to the task of verifying information flow policies. Before turning our attention to these, however, it should be recognized that access control may be an appropriate security

policy in its own right, at least for many less critical applications. In these cases access control verification provides all the assurance of security that is required. An example of this approach is provided by the UCLA Data Secure Unix System [POPE78b, POPE79, WALK80] which enforces `data security´: an access control policy which requires that user data may not be read or altered except in accordance with recorded `protection data´. This does not preclude user data being `leaked´ by covert means but is probably quite adequate for many non-military applications. And it should be noted that although its security policy may be rather limited, the UCLA system is probably the most completely verified of all current systems and is, indeed, one of the most realistic and substantial examples of program verification performed to date.

4. VERIFICATION BY INFORMATION FLOW ANALYSIS

The evidence of the previous section suggests that access control verification is unsuited to the task of proving that our toy system enforces the security policy of isolation. The problem, of course, is that the policy of isolation is concerned with information flow whereas access control deals only with physical access to the repositories of information. If we are to guarantee the absence of leakage channels, we need a verification technique that addresses squarely the issue of information flow. Before examining such a technique, however, it seems best to investigate the different types of leakage channel that may be present in a system.

Lampson, who first identified the threat of leakage channels [LAMP73] enumerated three different types. The first type, exemplified by the scenario given in the previous section, is the **storage** channel - it uses the kernel's own storage as its means of communication. In the example it was the kernel's record of the access allowed by B to U that was used. Second is the **legitimate** channel, in which illicit information is `piggybacked' onto legal information flow (by modulating message length, for example). The third of Lampson's types of leakage channel is the **timing** or **covert** channel which achieves communication by modulating some detectable aspect of the system's performance. Suppose, for example, that the colluding users A and B each have a pair of subprocesses, (A1,A2) and (B1,B2) respectively, where A1 and B1 are CPU-bound and A2 and B2 are I/O bound. If A chooses to execute A1 rather than A2, then the impact on the system's workload will be such that the scheduler will prefer to dispatch B2 rather than B1. If B can sense the rate at which his subprocesses are being serviced, then he can infer which of A1 or A2 is being run. Once again, B can sense activity by A and the basis of a communication channel has been established.

Storage and legitimate channels are considered a more serious threat than timing channels since they are potentially more numerous, less noisy, and of higher bandwidth. Fortunately, and unlike timing channels, their analysis is tractable. Lipner [LIPN75] and Millen [MILL76] were the first to argue that channels of the storage and legitimate varieties result from a failure to identify all the objects within the system in sufficient detail. Objects - the repositories of information - must be considered to include, not just files, but all storage locations within the system, including the kernel's own variables. It is necessary to assign appropriate security classifications to all kernel variables and to ensure that the kernel satisfies appropriate restrictions on its own accesses to these variables.

The necessary guarantees on the kernel's own behaviour must be obtained by compile-time analysis of its code. The purpose of this analysis is to detect all information flows from one kernel variable to another that can possibly occur during execution of the kernel. A technique for performing this `information flow analysis' (sometimes also called `security flow analysis') was first proposed by Denning [DENN75, DENN76, DENN77]. Denning's method was originally intended to "certify the secure execution of a program in an otherwise secure system" [DENN75, page 9] - in other words, to be applied to `ordinary' programs rather than security kernels. The extension of Denning's work to security kernel applications is largely due to Millen [MILL76, MILL79].

The basic idea of information flow analysis is to study the functional specification of a system in order to discover all potential information paths from one variable to another. This is accomplished by applying a set of `flow rules´ to the specification. For each construct of the specification (or programming) language concerned, these flow rules tell us whether it is possible for information to flow from one of its components to another. For example, in the specification language used for our toy system it is obvious that the construct

$$Y' = X$$

causes information to flow from X to Y. Less obvious, perhaps, is the fact that information can flow from variables appearing in **PRECONDITIONS** to those changed by **EFFECTS**. Consider, for example, the specification

```
OPERATION OP
PRECONDITIONS
    X ≠ 0
EFFECTS
    Y' = 1
END
```

Here, too, there is a potential information flow from X to Y - for after execution of OP, the value of Y can tell us something about the value of X. Clearly, if $Y \neq 1$ after execution of OP then we know for certain that its **PRECONDITIONS** could not have been satisfied and, therefore, that $X = 0$. If $Y = 1$, however, we cannot be so certain that $X \neq 0$ - it depends on what we know about the value of Y prior to the execution of OP. Defining precisely when information does flow from one variable to another is a difficult problem [BEST80, COHE78] but for our purposes the weaker notion of whether information can flow is sufficient. A set of rules for a specification language similar to that used here has been worked out by Millen [MILL76, MILL79]. These rules are quite complex, particularly with respect to subscripted variables (see also [DENN80, GOSS80]), but since I am only concerned with general principles, these complexities can be ignored here.

The important point is that the flow rules enable us to identify all potential information paths within the system. The next step is to determine whether any of these paths, or any combination of them, can provide users with the means to communicate with one another and thereby violate the security policy of isolation. Now the interface between our toy system and its users is provided by the I/O devices attached to the main memory and what we must prove, therefore, is that no combination of individual flow paths can constitute an information path between memory blocks attached to I/O devices of different colours.

One possible method for constructing such a proof is to assign a colour to each state variable and then to require that no individual information path connects variables of different colours. Induction then provides the desired conclusion that no combination of paths can lead to a security violation. Notice, however, that the assignment of colours to variables cannot be a purely static one. The whole problem of system security arises because different regimes share the same resources - and so the colour of a resource must change according to the colour of the regime currently controlling it. The general registers provide a simple example of this problem in our toy system. The colour

of the registers must change with the colour of the active regime: when a black regime has control, these registers must surely be coloured black; likewise when control passes to a red regime, the general registers must become red also. Thus the colour of a variable will, in general, be a function of the system state. The previous argument, that for overall security it is sufficient to establish that each individual information path connects only variables of the same colour, remains true - except that now we must evaluate the colour of the source variable in the system state which obtains prior to the execution of the operation, while the colour of the destination variable is evaluated in the state following its execution.

A subtle consequence of the fact that the colour of a variable may change during the execution of an operation is that the flow of information from a variable to itself now becomes significant: the value of a variable may be unaltered by an operation, but if the operation changes the colour that variable, then a potentially dangerous information flow has occurred even though the variable has not participated in any data flow.

A further complication arises from the fact that certain information (for example the fact that a particular memory block is currently `free´) may not be communicated to any regime (or else a fairly obvious security flaw opens up) and so variables recording this information cannot be given the colour of any regime. This is the reason for the introduction of the pseudo-colour `SYSTEM´ which behaves as a universal information sink: information of any colour may flow into it, but none may flow out. Conversely, certain information (system constants, for example) may pass freely to all regimes and may most conveniently be given the pseudo-colour `PUBLIC´ which fulfills a contrary role to SYSTEM: information may flow from it, but not to it. In effect, we have induced a partial order (denoted by $\leq$) over the enlarged colour set where

$$\text{PUBLIC} \leq c \leq \text{SYSTEM}$$

for each `real´ colour c, while distinct `real´ colours are incomparable. What we are now required to prove is that if any operation can cause information to flow from a variable X to a variable Y, then the colour of X (evaluated in the `old´ system state) stands in the relation $\leq$ to the colour of Y (evaluated in the `new´ system state). I shall call this partial order the **flow policy** and say that we require the information flows to **respect** it. (Notice that since the flow policy is a partial order, a general multilevel security policy can be handled as easily as one of isolation.)

Having explained the basic ideas behind information flow analysis, we shall now see how it works in practice with our toy system. First, we need to find an assignment of colours to the variables of the system. It is important to note that, apart from those variables which can be directly observed by the outside world and whose colours are therefore fixed by external considerations (in our case, this means those memory blocks which are attached to I/O devices), the assignment of colours to variables is, in a sense, arbitrary. We just need to find some assignment of colours for which the flow policy is respected.

Intuition suggests that it is sensible for those variables which record information about a memory block to be given the colour of that block, while machine registers are given the colour of the currently active regime and the saved copies of machine registers are given the colour of the regime on whose behalf they are saved. Thus we arrive at the tentative assignment given below. (Refer back to Figure 1 for an explanation of the use of each variable.)

VARIABLE	COLOUR
MEM(b,w)	BCOLOUR(b)
R(i)	RCOLOUR(AR)
MAP(j)	RCOLOUR(AR)
SR(r,i)	RCOLOUR(r)
SMAP(r,j)	RCOLOUR(r)
RCOLOUR(r)	PUBLIC
BCOLOUR(b)	BCOLOUR(b)
ACCESSIBLE(b)	BCOLOUR(b)
FIXED(b)	PUBLIC
AR	PUBLIC

We now have to see whether, with this assignment of colours to variables, the information flows of each operation respect the flow policy.

To take an easy case first, the operation LOAD(i,j,w) can be seen to generate a direct flow from MEM(MAP(j),w) to R(i). The colours assigned to these variables are BCOLOUR(MAP(j)) and RCOLOUR(AR) respectively and so we must require

$$(\forall j)\ \text{BCOLOUR}(\text{MAP}(j)) \leq \text{RCOLOUR}(\text{AR}).$$

As it happens, this relation is an immediate consequence of the access control policy (*) proved (or, rather, assumed proved) earlier and so this operation certainly respects the flow policy.

Now for a more difficult case. The operation ACQUIRE(b) does not affect the value of ACCESSIBLE(b) - but that means that it generates a flow from ACCESSIBLE(b) to itself! (Recall the discussion on the previous page.) The colour of ACCESSIBLE(b) is given by the value of BCOLOUR(b) - which can be altered by the ACQUIRE(b) operation. If the prior value of BCOLOUR(b) is SYSTEM, then its value after the operation is RCOLOUR(AR). Therefore, the colour of the information in ACCESSIBLE(b) can change from SYSTEM to RCOLOUR(AR) and so we must require

$$\text{SYSTEM} \leq \text{RCOLOUR}(\text{AR}).$$

But this cannot possibly be true, since RCOLOUR(AR) is a `real' colour.

This does not necessarily imply that the system is insecure, however, because a failure to respect the flow policy can arise for reasons other than insecurity. In the first place, it may simply be that the assignment of colours to variables is inappropriate: the flow under

consideration may respect the flow policy if a different assignment of colours is chosen. If this is the case, then we may simply substitute the new assignment for the old (after rechecking all the flows examined previously) and proceed. Millen [MILL79] states that choosing a good assignment requires a combination of experience, guesswork and trial and error.

In the case of the troublesome flow in ACQUIRE, however, no reassignment of colours to variables seems to solve the problem - but still this does not necessarily indicate an insecurity. This is because the flow rules are conservative: they do not tell us whether there is a flow from X to Y, but whether it is possible that there could be such a flow. Furthermore, the flow rules are syntactic: they are sensitive to the way a specification is written and a flow may vanish when a specification is rewritten in a syntactically different, yet semantically equivalent manner. Millen [MILL78, MILL79] calls these apparent security violations `formal´ since they arise from the form of the specification, not from any real information channel.

It turns out that the flow, in ACQUIRE, from ACCESSIBLE(b) to itself is formal: it may be removed by the rather drastic expedient of deleting that entire array of state variables from the system. This is possible because ACCESSIBLE(b) is redundant - the expression

not ACCESSIBLE(b)

which appears in the **PRECONDITIONS** for ATTACH and RELEASE can be replaced by the universally quantified expression:

$$(\forall j)\ \mathrm{MAP}(j) \neq b$$

Unfortunately, the flow involving ACCESSIBLE is not the only troublesome one in ACQUIRE. There is also a flow from BCOLOUR(b) (in the **PRECONDITIONS**) to BCOLOUR(b) (in the **EFFECTS**). Since the colour of BCOLOUR(b) is its own value - which, as we have seen, may start off as SYSTEM and finish as RCOLOUR(AR) - we again require the impossible relationship.

$$\mathrm{SYSTEM} \leq \mathrm{RCOLOUR(AR)}.$$

No reassignment of colours can remove this problem, nor does the flow appear to be formal - so perhaps we have found a real security flaw after all. The only way to convince ourselves of this is to construct a scenario which actually demonstrates a security compromise.

It is fairly easy to construct a suitable scenario for the present case: it depends upon the fact that memory blocks are explicitly named in the ATTACH, ACQUIRE and RELEASE operations.

Suppose the currently active regime performs the three operations ACQUIRE(0), ATTACH(0,0) and LOAD(0,0,0). (That is, it first requests ownership of block number 0, then attaches that block to mapping register 0, and finally, it loads the contents of the first word of the block attached to mapping register 0 into general register 0.) Then there are two possible outcomes: if block 0 was formerly free (i.e. BCOLOUR(0) = SYSTEM before the ACQUIRE) then it will be given the colour of the currently active regime and be made accessible to that regime

through mapping register 0. The final contents of register R(0) will then be equal to zero - since they are loaded from word 0 of block 0 which was zeroed when the block was ACQUIREd. On the other hand, if block 0 is currently owned by some other regime, then the ACQUIRE(0) and ATTACH(0,0) operations have no effect (since their **PRECONDITIONS** will not be satisfied) and the memory block accessible through mapping register 0 will be not block 0 but whichever block was accessible through that register previously - say block b. If the contents of MEM(b,0) were earlier set to some non-zero value, then the final value in R(0) will be non-zero also. Thus R(0) is set to zero or non-zero according to whether block 0 is free or not. This means that another regime can signal to the current one by ACQUIRing and RELEASing block 0 and so, because there is a communication channel (albeit a noisy one) from one regime to another, the system is insecure.

There is no simple way to remove this flaw from the system since it is due to the fact that memory blocks are subject to a global naming scheme: a regime can request that a particular block be ACQUIREd and can subsequently detect whether this operation was performed successfully or not. A plausible repair would be to allow only that some block be ACQUIREd. But this operation must fail when no free blocks remain and so one regime can still signal to another by controlling the availability of free blocks. (The technique of suspending a regime until a block becomes available does not solve the problem either; it merely replaces a storage channel by a timing channel - and also admits denial of service and deadlock problems.) The existence of any finite shared resource must always cause a security flaw of this type unless each regime has its own individual resource quotas and the system is able to honour all of these quotas simultaneously. In the case of our toy system, this could be accomplished by allowing each regime to own no more than MAX memory blocks, where MAX * n $\leq$ 128 (remember, there are n regimes and 128 memory blocks in total). In this case, there is no dynamic memory sharing and so no point in providing the RELEASE and ACQUIRE operations - we may as well fix the memory blocks to be allocated to each regime once and for all at system initialisation time. The resulting system can be made a little more interesting by allowing the use of virtual, rather than actual block numbers in the ATTACH operation (i.e. each regime numbers its blocks 0,1, ... ,MAX-1.) This modified system is specified in Figure 2. The new array called ACTUAL is introduced to perform the mapping of virtual to absolute block numbers: ACTUAL(r,v) contains the actual block number of the r'th regime's virtual block number v. (Henceforth, the symbol v is bound to the range 0 $\leq$ v < MAX). Naturally, we require that ACTUAL(r1,v1) = ACTUAL(r2,v2) only if r1 = r2 and v1 = v2. Since there is no dynamic memory sharing, the state variables RCOLOUR, BCOLOUR, ACCESSIBLE, and FIXED are no longer required.

Insofar as it led us to discover the insecurity in the toy system of Figure 1, the method of information flow analysis seems an effective technique. But what of the positive aspects of verification? Suppose our analysis revealed no unpleasant flows - could we then be sure that the system was secure? The answer is that we cannot. The reason is that so far we have only examined information flow caused by the operations themselves; we have not considered the possibility of information flow caused by the ability to choose which operations to execute. That may sound a little enigmatic so let me explain. Operations which move data from a `red' variable to a `black' one are obviously insecure - and the techniques introduced so far will detect this kind of insecurity.

```
OPERATION LOAD(i,j,w)
EFFECTS
    R(i)´ = MEM(MAP(j),w)
END

OPERATION STORE(i,j,w)
EFFECTS
    MEM(MAP(j),w)´ = R(i)
END

OPERATION ATTACH(v,j)
EFFECTS
    MAP(j)´ = ACTUAL(AR,v)
END

OPERATION SWAP
EFFECTS
    AR´ = AR+1 (mod n)
    (∀i) R(i)´ = SR(AR´,i)
    (∀i) SR(AR,i)´ = R(i)
    (∀j) MAP(j)´ = SMAP(AR´,j)
    (∀j) SMAP(AR,j)´ = MAP(j)
END
```

Figure 2 : Modified Kernel Specification

But now consider an operation which moves data from one `red´ variable to another. This operation seems safe enough - and so it is provided it is executed by a `red´ regime; if a `black´ regime has the ability to execute this operation then, by choosing whether to exercise that ability or not, it has the power to decide whether the `red´ destination variable will be altered or not - and consequently the power to communicate information to the `red´ regime. Thus, the execution of an operation can cause information to flow from the active regime to all those variables which may be altered by the operation. We can ensure that this information flow is safe by requiring that each variable which may be altered by an operation has the same colour as the active regime. Feiertag et al. [FEIE77] have shown that this extended form of information flow analysis is sound:* that is, if a system is shown to be secure by this techniques, then it is truly secure.

Encouraged by this guarantee of the soundness of the extended form of information flow analysis, we might now attempt to apply the technique in order to verify the security of the system specified in Figure 2. A fairly straightforward analysis shows that all the information flows caused by the operations LOAD, STORE and ATTACH are secure, but what of the SWAP operation? It is the essence of this operation that it

* More accurately, [FEIE77] establishes the soundness of information flow analysis for those systems whose behaviour is adequately described by the computational model used therein. In Section 6, I shall argue that this model does not capture all the salient characteristics of a security kernel and that information flow analysis does not provide a completely sound technique for verifying security kernels.

should save the register contents of the outgoing (say `black´) regime and restore those of the incoming (say `red´) regime. Thus it is essential that this operation should read and write both `red´ and `black´ variables. But, as we have just seen, a `black´ operation (i.e. one invoked by a `black´ regime) cannot be allowed to write `red´ variables. It follows that the method of security verification by information flow analysis is unable to attest to the security of the SWAP operation and so fails to verify the security of the system given in Figure 2 (even though this system is secure).

This inability to verify a truly secure system would not be so bad if the troublesome SWAP operation could be considered a pathological or a special case: we could then construct an ad-hoc argument for this one particular circumstance and rely on information flow analysis elsewhere. However, I do not consider this a tenable position. Rather, I believe that operations like SWAP are right at the heart of the problem of security kernel verification - for one of the fundamental tasks of a security kernel is to allow the limited physical resources of the computer system to be shared securely among a number of regimes. The operation SWAP is the means by which the kernel allows for the secure sharing of the CPU resource. In a real system there will be many other resources to be managed, and many of the operations upon those resources will surely require simultaneous access to components of different `colours´. Operations involved with paging, shared I/O devices and backing store are all likely to have this character. Far from being a pathological case, SWAP is the very paradigm of those operations which embody the fundamental character of a security kernel.

If information flow analysis is to provide a sound basis for verifying the security of operations such as SWAP, then it is clear that we must seek modifications or extensions to the method. Simple modifications or extensions will not suffice, however, because information flow verification is based on the assumption that a system can be verified by considering one operation at a time. This assumption is not universally valid. To see this, recall the system of Figure 2. This system is secure. Now consider a system in which the SWAP operation of Figure 2 is replaced by the operation NEWSWAP defined as follows:

```
OPERATION NEWSWAP
EFFECTS
    AR´ = AR+1 (mod n)
    (∀i) R(i)´ = 0
    (∀j) MAP(j)´ = SMAP(AR´,j)
    (∀j) SMAP(AR,j)´ = MAP(j)
END
```

(NEWSWAP is just like SWAP except that instead of saving and restoring the general registers, it zeros them.) This is also a secure system. Both the system with SWAP and that with NEWSWAP are secure because, in each case, a regime knows, when it gives up control, exactly what values its general registers will contain when next it receives control back again. Now suppose both SWAP and NEWSWAP are available. This system is manifestly insecure since a regime can signal to its successor by choosing whether to zero the general registers of the incoming regime (using NEWSWAP) or to restore their previous values (using SWAP). But which is the insecure operation - SWAP or NEWSWAP? Of course it is neither individually: it is the presence of both which is insecure.

It follows that a security verification technique cannot consider each operation in isolation but must consider the system in its totality. Furthermore, it is often necessary to consider the actual values possessed by kernel variables, not just their `colour´. (This is the key to verifying the SWAP operation - we must prove that it restores the same register contents as it previously saved.*) These are such radical departures from the philosophy implicit in information flow analysis that the conclusion seems inescapable: that technique, on its own, is inadequate to the task of security kernel verification. It must either be replaced or reinforced by some quite different technique.

This conclusion is not a new one. In its application to security kernels, information flow analysis is conventionally seen as a technique to be applied to the high level `design specifications´ [BERS79, McCA79a]; the task of demonstrating that the actual implementation is secure is then accomplished by proving it to be a correct implementation of these secure specifications. If this two-stage approach is sound (and in Sections 6 and 7 I shall argue that it is not), then it can only be so if both its stages are actually carried out. In conventional practice, however, the second stage is not performed: the KSOS contract, for example, called for only `illustrative´ proofs of the implementation [BERS79].

The explanation for this failure to perform the second stage of verification concerns cost and perceived risk: proving the correctness of an implementation with respect to its specification is a far more complex and costly procedure than performing information flow analysis on the specification - while its apparent benefit is considerably less since it is tacitly assumed that all the security-critical aspects of the design are embodied in the design specification and that the transition from this specification to the actual implementation is merely a coding exercise. If this coding is performed by skilled and trusted personnel and subjected to rigorous, though informal, checks, then surely, it might be argued, the risk of introducing a security flaw into a proven secure design is sufficiently small that the cost of full correctness verification is unjustified. This analysis may seem plausible but is, in my view specious.

Firstly, it underestimates the risk of introducing a security flaw during the elaboration of the high-level specification into one at implementation level - for this elaboration involves, not mere coding, but the introduction of design decisions that are crucial to security. In the case of the toy kernel of Figure 2, for example, the description of the SWAP operation would be removed from the specification to the implementation level if this two-stage approach were followed. Verifying the security of the high-level specification would then become an affirmation of the obvious, while the crucial issue of determining the security of the SWAP operation remained unbroached. (I shall work this example through in some detail in Section 6.)

---

* Andrews and Reitman [ANDR80] have developed a technique that combines flow analysis with correctness verification. Using their method it may be possible to prove, for example, that SWAP restores the same register values as it saved. However, this does not solve the really crucial problem: how do we deduce that this is the property of SWAP that is vital for security?

In the case of large, complex systems such as KSOS, where the kernel contains, among other things, the mechanism to support a multilevel secure file system, there are so many opportunities for security flaws in the design specifications that threats from the implementation level may seem small by comparison. But while they may be small in comparison with the threats from elsewhere, these threats are not small in absolute terms. The systems with which I have been personally concerned [BARN80] are sufficiently simple that the security of their high-level description is obvious. All the perceived threat in these cases comes from the implementation stage. And when this stage is examined in isolation, the threat is seen to be considerable - for the security of the whole enterprise rests on the manipulation of some of the nastiest and trickiest aspects of the underlying hardware, notably the memory management facilities, the I/O system, and the interrupt mechanism. It is precisely in the secure management of such complex details as these that the guidance and reassurance of an appropriate verification technique is vital. Indeed, Popek's group have observed:

> "One should realise that it is essential to carry the verification process through the steps of actual code-level proofs because most security flaws in real systems are found at this level" [WALK80, p118]

Not only does the conventional analysis underestimate security threats due to flaws in the implementation rather than the specification but also, by assuming that the absence of such flaws can be guaranteed by `ordinary' correctness verification, it fails to cast any light on the problem of just what it is that makes an implementation secure. Clearly it is somewhat extravagant, when all we require is an assurance of security, that we should have to prove utter correctness, but it is not simply on aesthetic grounds, nor even on those of cost, that I object to this approach to verifying the security of an implementation; my concern is that it indicates a failure to isolate the significant issues: if we cannot separate the security of an implementation from other aspects of its correctness, then, surely, we do not have a really firm grasp on this concept of `security'.

A verification technique is not simply a procedure to be applied blindly, pronouncing `correct' or `incorrect' as the case may be; much more important is the insight that its mastery provides. Familiarity with the idea of a `loop invariant', for example, benefits the understanding of all who write programs, not only those who must prove them. The discipline and insight engendered by a verification-oriented approach to system design and implementation not only allows us to prove our programs correct, it helps us get them correct in the first place. To prove the security of an implementation by way of its total correctness is to use a rather blunt instrument; a technique specific to security could assist in the separation of concerns and provide guidelines for the systematic construction of secure implementations.

It is to the development of just such a technique that I now wish to turn.

5. VERIFICATION BY PROOF OF SEPARABILITY

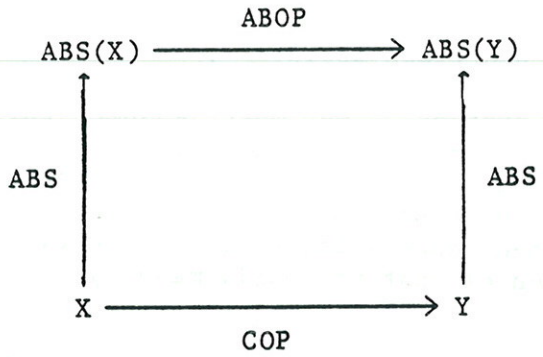
Some security problems only arise when different regimes share a single system. Red/black isolation, for example, ceases to be an issue when the red and black regimes each run in their own private and physically separate systems. We must assume that external constraints (presumably of cost) prevent the adoption of this simple expedient and instead require all regimes to share a single system. But if this shared system exhibits precisely the same behaviour as the multiple-system ideal then surely it, too, is secure. Indeed, is this not the natural definition of what we really mean by a security policy of isolation?

It follows that a plausible method for verifying the security of a system is to prove that, from the viewpoint of each individual regime, the visible behaviour of the shared system is indistinguishable from that which would be provided by some private, unshared system dedicated to that regime alone. Notice that it is not necessary to exhibit such a private system; it is sufficient merely to prove that one could exist. However, there are additional benefits to be gained from a more constructive approach in which a description of an idealised, private system is actually produced for each regime. Not the least of these benefits is the fact that the specifications of the idealised systems provide a convenient and exact description of the kernel interface seen by each regime. I shall call these idealised systems `abstract', as distinct from the `concrete' system which is shared by all regimes and whose security is to be verified.

Given the specifications of the abstract systems, verification of the security of the concrete, shared system can be achieved by proving that its behaviour is `consistent' with each of these private, abstract systems. We now need a precise interpretation of what we mean by `consistent' and a method for establishing the presence of this property. In this regard, it will prove useful to digress from the discussion of security for a short while in order to recall some of the issues concerned with the related problem of verifying the correctness of implementations of abstract data types.

An abstract data type consists of a state together with a collection of operations for changing that state and for extracting information from it. The specifications of an abstract data type form a succinct, high level description of its behaviour and define an interface between those parts of a system which implement the type and those which use it. The basic method for proving that the implementation of a type satisfies its specifications is due to Hoare [HOAR72] and has subsequently been further developed by others, notably the Alphard Group [WULF76]. In this method, the states and operations defined by the specifications are described as **abstract** states and operations while those of the implementation are described as **concrete.** The key idea (due to Milner [MILN71]) is to relate these two levels by means of an **abstraction function.** This is a function, called ABS say, from concrete to abstract states: ABS(X) is the abstract state represented by the concrete state X. Note that abstraction functions are, in general, many to one (several different concrete states may all represent the same abstract state) and partial (some concrete states may not represent any abstract state at all). For an implementation to be correct with respect to its specification, we require that the following diagram

commutes (where ABOP is the abstract operation which is implemented by the concrete operation COP):

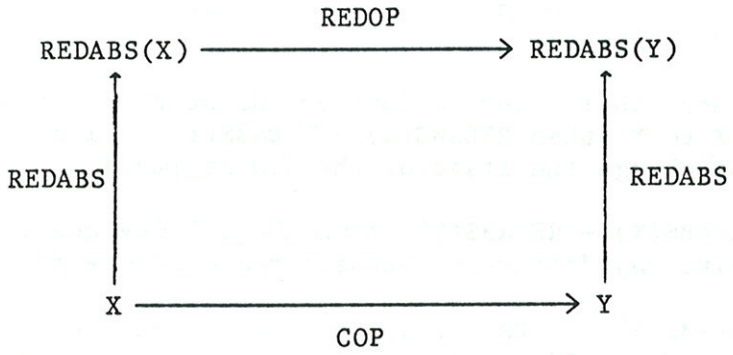


That is to say, if the concrete system is in the state X, representing the abstract state ABS(X), then the new abstract state obtained by applying the abstract operation ABOP to ABS(X) should be the same as that represented by the concrete state Y which results from applying the concrete operation COP to X. This approach is treated more carefully in [WULF76], where further conditions (involving the use of abstract and concrete `invariants´) are placed upon the notion of correctness, but the idea of an abstraction function remains central to all later refinements since it provides the connection between the concrete and abstract views of the system.

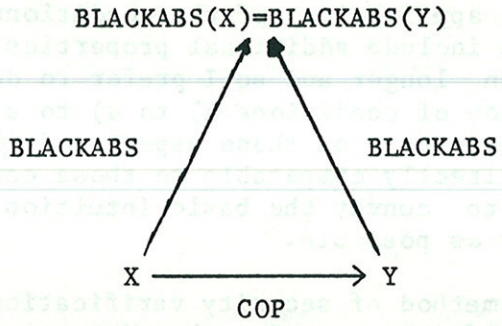
With these ideas in mind, let us now return to the consideration of secure systems. Here we had distinguished a concrete version of the system - which is shared by all regimes, and a set of abstract versions - each one private and dedicated to a particular regime. There is a similarity to be found here between the concrete and abstract views of a secure system and those of an abstract data type. Perhaps then, we can draw on this similarity and attempt to carry the idea of an abstraction function over to secure systems. In this way it may be possible to reconcile the concrete view of a shared system with the abstract view of a private system and to define a notion of `consistency´ between them. Notice, however, that unlike the case of an abstract data type, where an implementation is required to be consistent with one abstract specification, a concrete secure system is required to support multiple abstractions simultaneously - one for each regime. The key to our new security verification technique is to handle this multiplicity of abstractions in the most natural way possible - by introducing a multiplicity of abstraction functions. In the simple case of a red/black system, for example, we will have two such functions: REDABS and BLACKABS. The former will map states of the concrete shared system into those of the red regime´s abstract private system, while the latter does the same for black.

I am now going to restrict my attention to those concrete systems which `time share´ between their different regimes. That is systems (such as those of Figures 1 and 2) in which operations are performed at the behest of one currently `active´ regime. (All systems likely to arise in practice are of this type.) Suppose it is the red regime that is active when the system performs a concrete operation COP. Then we must require that the effects of this operation, as perceived by the red regime, are just as if some operation REDOP has been executed by the red abstract system. Thus, if execution of COP takes the concrete system

from an initial state X to a final state Y, we demand that REDABS(Y) is the same state of the red abstract system as that which results from applying the abstract operation REDOP to the abstract state REDABS(X). In other words, just as with an abstract data type, we require the following diagram to commute:



This condition ensures that the active regime cannot distinguish between the behaviour of operations performed on its behalf by the concrete system and those which would be performed by its own abstract system. 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 system perceived by inactive regimes. For secure isolation we must, therefore, also require that the transition between concrete states X and Y produces no corresponding change in the states of abstract systems belonging to inactive regimes. That is, in the case of the black regime, we require that BLACKABS(X) = BLACKABS(Y), or in diagrammatic form:



If these were the only conditions attendant upon the operations and the abstraction functions, then all systems could be `proved´ secure by the trick of taking every abstract system to consist of but a single state which is preserved by all operations. (That is, a system which never does anything.) To discover what additional conditions are needed, let us recall the intuition we are seeking to formalise in this method of security verification. The basic idea is that, to each of its users, the shared system is to be indistinguishable from another, unshared one. If the shared and the unshared systems were hidden behind a screen, the user must be unable to determine which of them is actually connected to his I/O devices. And this raises the point that we have missed until now: the two systems must be `I/O compatible´. The abstract system cannot be chosen arbitrarily; it must be sufficiently `realistic´ that it can simulate the I/O behaviour (and instruction sequencing mechanism) of the concrete system.

Informally (and expressed solely in terms of the RED regime), the additional conditions required are:

a) If activity by a RED I/O device changes the state of the concrete system from X to X´, and the same activity changes state Y to Y´, then REDABS(X) = REDABS(Y) must imply 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 a non-RED I/O device changes the state of the concrete system 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 RED I/O devices must not be able to perceive any difference between the concrete states X and Y.

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. A precise statement of all these conditions requires rather more formalism than I am willing to introduce here, although a complete and rigorous formal development of the method can be found in [RUSH81b]. More serious, perhaps, than the lack of formality in the statement of these properties is the absence of any illustration of their practical interpretation and significance. The toy kernel specification which I have been using to illustrate the various techniques does not contain any description of the behaviour of I/O devices or the instruction sequencing mechanism. Consequently, it cannot be used to illustrate the application of the conditions a) to d). Extending the specification to include additional properties would make this already long paper even longer and so I prefer to defer detailed illustration and interpretation of conditions a) to d) to a subsequent report. Here I want to concentrate on those aspects of this new verification technique that are directly comparable to those considered already. I hope, in this way, to convey the basic intuition and motivation behind the method as simply as possible.

Since this method of security verification relies on showing that the behaviour of the single, shared system is indistinguishable from that of a collection of physically separate, unshared systems, I shall refer to it as verification by **proof of separability**. This name may not be succinct, but it is at least mnemonic. By way of an example of the use of the technique, I will sketch a proof of the security of the system which was specified in Figure 2.

First we need a specification of the abstract system perceived by each regime. In this case each regime sees a different copy of the same abstract system and so a single specification (or perhaps it is better called a specification schema) will suffice. This specification is given in Figure 3. Notice that the operations and state variables of the r´th regime´s abstract system are distinguished by a subscripted r. Unsubscripted operations and state variables continue to refer to the concrete system of Figure 2. It is instructive to compare the operations of Figures 2 and 3. Note, in particular, that SWAP is perceived

State Variables

$MEM_r(v,w)$ word w of block v of main memory

$R_r(i)$ i´th general register

$MAP_r(j)$ j´th mapping register

Operations

OPERATION $LOAD_r(i,j,w)$
EFFECTS
$R_r(i)' = MEM_r(MAP_r(j),w)$
END

OPERATION $STORE_r(i,j,w)$
EFFECTS
$MEM_r(MAP_r(j),w)' = R_r(i)$
END

OPERATION $SWAP_r$
EFFECTS
{none}
END

OPERATION $ATTACH_r(v,j)$
EFFECTS
$MAP_r(j)' = v$
END

Figure 3 : Kernel Specification of `Abstract´ System

as a no-op from each regime´s viewpoint. (The perceived **EFFECTS** of NEWSWAP would be $(\forall i)\ R_r(i)' = 0$.)

Next we need to construct the abstraction functions. The abstraction function for regime r is a mapping from the state space of the concrete system to that of the r´th regime´s abstract system. Since these state spaces have been defined implicitly (in terms of variables and the effects of the operations upon them) rather than explicitly (as sets), it is inconvenient to define the abstraction functions explicitly; instead, following [ROBI77] (for example), I will define them implicitly by asserting the following equivalences between expressions involving abstract and concrete state variables:*

$$MEM_r(v,w) = MEM(ACTUAL(r,v),w) \quad (1)$$

$$R_r(i) = \textbf{if } r=AR \textbf{ then } R(i) \textbf{ else } SR(r,i) \quad (2)$$

$$ACTUAL(r,MAP_r(j)) = \textbf{if } r=AR \textbf{ then } MAP(j) \textbf{ else } SMAP(r,j) \quad (3)$$

These equations express how the variables of the abstract systems are

* Actually, this problem of defining abstraction functions raises some

represented within the concrete system. Consequently, I shall speak of each right hand side as the **representation** of its corresponding left hand side. In the case of (2) and (3), where the equations have the form:

abstract_expr = **if** r=AR **then** concrete_expr$_1$ **else** concrete_expr$_2$

I shall speak of concrete_expression$_1$ as the **active** representation of the abstract_expression on the left hand side and concrete_expression$_2$ as its **passive** representation. The right hand side of (1) is both the active and the passive representation of its left hand side.

We now need to use these representations in order to establish consistency between the operations of the concrete and abstract systems of Figures 2 and 3. Consider the effect of the abstract operation $LOAD_r(i,j,w)$ upon the abstract variable $R_r(i)$. From Figure 3, we see that the new value of $R_r(i)$ is given by the abstract expression

$$MEM_r(MAP_r(j),w). \qquad (4)$$

Now the effect of $LOAD_r(i,j,w)$ upon $R_r(i)$ should be consistent with the effect of the corresponding concrete operation LOAD(i,j,w) upon the representation of $R_r(i)$. Since the concrete system only executes operations on behalf of the active regime, we must have r = AR and so it is only the active representation of $R_r(i)$, namely R(i) that we need to consider. Figure 2 tells us that the new value of R(i) after execution of LOAD(i,j,w) is

MEM(MAP(j),w)

and we now need to show that this is, indeed, a valid (active)

---

fascinating and rather deep issues. In formal treatments (for example [GOGU78]), an abstract data type is regarded as a (many sorted) algebra. The notion of correctness which I described earlier (in terms of abstraction functions) then corresponds to a homomorphism on algebras. However, we have not explicitly constructed the algebras corresponding to our abstract and concrete systems, we have merely given specifications of them. Technically, our specifications are `theories´ (or would be, if we were doing this formally) and the systems (algebras) which they specify are the models (in the logicians´ sense) of those theories. Since we are working with theories, rather than the algebras they specify, we really need a notion of `correctness´ that operates at this level. Nakajima et al. [NAKA78, NAKA80] argue that the logicians´ notion of **interpretation** [ENDE72, SHOE69] is appropriate for this purpose. An interpretation is defined by a translation from the terms of the abstract specification into those of the concrete one. (As opposed to an abstraction function which is a homomorphism from the concrete to the abstract algebra.) For correctness, we require that when the axioms of the abstract theory are translated into the language of the concrete theory, they become theorems of that theory. Although I have explained and informally justified `proof of separability´ in an implicitly `algebraic´ setting, the example of its application employs a `logical´ approach (with equations (1), (2), and (3) defining the translation from abstract to concrete terms).

representation of the abstract expression (4). This is easily done, for (1) gives

$$MEM_r(MAP_r(j),w) = MEM(ACTUAL(r,MAP_r(j),w))$$

and since (3) gives (in the case that r = AR)

$$ACTUAL(r,MAP_r(j),w) = MAP(j)$$

we deduce

$$MEM_r(MAP_r(j),w) = MEM(MAP(j),w)$$

as required.

What we have done here is to show that the same result is obtained independently of whether we first apply an abstract operation and then take the concrete representation of the abstract result, or take the representation first and then apply the corresponding concrete operation. Readers who are familiar with the notion of rewrite rules [HUET80b] may care to note that this procedure for verifying the consistency of $LOAD_r$ and LOAD with respect to their effects on $R_r(i)$ is equivalent to showing that both sides of the equation

$$R_r(i) = MEM_r(MAP_r(j),w) \quad \text{(from Figure 3)}$$

have the same `reduced form´ under the the following set of rewrite rules:

$R(i) \rightarrow MEM(MAP(j),w).$ (from Figure 2)

$MEM_r(v,w) \rightarrow MEM(ACTUAL(r,v),w)$ (from 1)

$R_r(i) \rightarrow R(i)$ (from 2)

$ACTUAL(r,MAP_r(j)) \rightarrow MAP(j)$ (from 3)

If course, this presupposes that the rewriting system has the `Church-Rosser property´ [ROSE73] (also called `unique termination´ [MUSS80a] and equivalent to `confluence´ [HUET80a]) but it raises the hope that parts of `proof of separability´ can be automated quite easily. (The Knuth-Bendix Algorithm [KNUT70] and its extensions [HUET80a, HUET80c, MUSS80b, PETE81] are relevant here.)

Similar arguments may be repeated for the other operations and state variables of the r´th regime´s abstract system in order to establish that, in each case, the behaviour of the concrete and abstract systems are consistent. The details, at least in the case of concrete operations which do not alter the identity of the active regime (i.e. all except SWAP) are straightforward, if a little tedious, since it is necessary to consider only the active representation of each abstract variable. SWAP is a little different because it does change the active regime and so it is necessary to consider both active and passive representations. Specifically, since the abstract effect of SWAP (i.e. $SWAP_r$) is a no-op, we need to show, for each abstract variable of the

`outgoing´ regime, that SWAP converts its active representation into its passive one. For example, since the active representation of $R_r(i)$ is $R(i)$ and its passive representation is $SR(r,i)$, it is necessary to ensure that SWAP contains the transition

$$SR(r,i)' = R(i)$$

- which indeed it does.

These arguments (suitably formalised) establish a consistency between the operations of the abstract and concrete systems. We also need to prove that the execution of a concrete operation has no effect upon the abstract variables of inactive regimes. This is elementary in the case of the three operations which do not alter the identity of the active regime since, in these cases, it is sufficient merely to note that concrete variables involved in the passive representation of abstract variables are never altered by these operations. In the case of SWAP, it is necessary to prove that the representation of each abstract variable of the `incoming´ regime is converted from its passive to its active form, while the passive representations of the variables of other regimes remain undisturbed.

All of these steps are conceptually straightforward and easily performed, at least informally. Reliable verification of realistic systems will require more formalised arguments and, almost certainly, mechanical assistance. Developments in these areas should follow once greater experience has been gained in the practical application of this verification technique. It is also possible that there could be some useful interaction between the techniques proposed here and those used in the verification of concurrent programs. (For example [OWIC79], requires that "each operation of a shared type must be described in terms of its effects on variables of the process invoking the operation".)

Before moving on to other topics, it is instructive to reconsider the original (insecure) system that was specified in Figure 1. If we were to attempt to verify the security of this system using the new technique of verification by proof of separability, we should have to begin by constructing specifications for each regime's abstract system. This means that from the single, composite system defined in Figure 1, we should have to try and disentangle the private view of the system perceived by each regime. Immediately we see that this puts the cart before the horse: for it must be more natural to begin with the desired private abstractions and then to use these to guide the construction of a concrete system which is formed as a synthesis of the individual abstractions. Indeed, it seems possible that large parts of this synthesis can be performed semi-mechanically once the concrete representations of the abstract variables have been chosen. So, unlike other techniques, such as information flow analysis, proof of separability not only provides a technique for verifying the security of systems, it also suggests and encourages a systematic methodology for the actual construction of such systems. Gross insecurities of the type present in the system specified by Figure 1 just could not occur using this methodology - for the insecurity in Figure 1 is manifest in the fact that it is impossible to construct any description of the behaviour of the system, as perceived by an individual regime, without making reference to objects external to that regime (or else introducing non-deterministic operations).

In conclusion, I claim that proof of separability provides a sound and natural technique for verifying the security of those kernelized systems which enforce a policy of isolation. In Section 7, I will show that its application extends to more complex policies also, but first I want to examine its relation to verification by information flow analysis.

6. INFORMATION FLOW ANALYSIS - REVISITED

We saw, in Section 4, that security kernels which enforce a policy of isolation cannot be verified by information flow analysis alone. Section 5 introduced the idea of `proof of separability´ and argued that this does provide a sound and practicable method for verifying these systems. In this present section I want to re-examine the role of information flow analysis in order to determine just what it is that this method of analysis can do for us. I will illustrate the two-stage approach to security kernel verification, in which information flow analysis is used to verify the security of a high-level `design specification´ while ordinary `correctness´ verification is used to prove the security of its implementation. I shall argue, however, that the contribution made by information flow analysis to the overall proof is almost negligible and the main burden falls to the other component - that is, to the correctness verification. I will then show that this correctness verification is most conveniently organised in such a way that it becomes nothing other than a variant of `Proof of Separability´. Finally, I will discuss the realism of the model used to justify the soundness of information flow analysis as a security verification technique [FEIE77] and will argue that, unlike proof of separability, information flow analysis provides no basis for verifying crucial security properties related to the flow of control, I/O devices, and interrupts.

Recall that the attempt to verify the security of the system specified in Figure 2 by means of information flow analysis foundered on the `colour change´ performed on the general and mapping registers by the SWAP operation. As D.E. and P.J. Denning say [DENN79, p237]:

> "We do not know how to certify programs that use variables whose security classes can change during execution."

In other words, methods based on information flow analysis cannot verify the secure use of shared objects. If there were no shared objects, then this problem would not arise. Why not postulate, therefore, a system in which there are no shared objects - that is, one in which each regime has its own set of general and mapping registers? This is the idea behind the two-stage approach to security kernel verification. Information flow analysis is not applied to a specification of the kernel implementation, but to a higher level `design specification´ from which all difficulties caused by the use of shared objects have been abstracted away. As Feiertag, Levitt and Robinson put it [FEIE77, p62]:

> "with respect to specifications, there is a separate machine for each security level."

The security of such a system can be verified by information flow analysis but we are then faced with the difficulty that this verifiably secure system (I shall call it a partitioned system) is not the same as the implementation we actually have available. We can overcome this difficulty, however, and thereby establish the security of the implementation, by proving that it exhibits exactly the same behaviour as the partitioned system. In conventional parlance, this is proving the **correctness** of the implementation with respect to the specification embodied in the partitioned system.

To make things definite, let us actually construct the `design specifications´ for a partitioned system corresponding to Figure 2. Rather than just one copy of each shared variable, we will now require an array of copies: one for each regime. For example, instead the single set of general registers R(i), each regime will now have its own set: those belonging to regime r will be denoted R(r,i). The state variables of the partitioned system are, therefore:

MEM(r,b,w)	word w of block b of regime r´s main memory
R(r,i)	i´th general register belonging to regime r
MAP(r,j)	j´th mapping register belonging to regime r
ACTUAL(r,v)	real block number corresponding to virtual block number v for regime r
AR	identity of the active regime

The operations of the partitioned system are shown in Figure 4.

```
OPERATION LOAD(i,j,w)
EFFECTS
    R(AR,i)´ = MEM(AR,MAP(AR,j),w)
END

OPERATION STORE(i,j,w)
EFFECTS
    MEM(AR,MAP(AR,j),w)´ = R(AR,i)
END

OPERATION ATTACH(v,j)
EFFECTS
    MAP(AR,j)´ = ACTUAL(AR,v)
END

OPERATION SWAP
EFFECTS
    AR´ = AR+1 (mod n)
END
```

Figure 4 : Specification of `Partitioned´ System

We can easily verify the security of this system by information flow analysis. The variable AR is coloured SYSTEM while all others are given the colour of the regime to which they belong: that is, MEM(r,b,w), R(r,i) and MAP(r,j) are all given the colour of regime r. We then observe that each operation specified in Figure 4 only generates information flow to and from the active regime. (Except SWAP which, because it changes the value of AR, also generates a flow to the `superior´ pseudo-colour SYSTEM.) All these flows are safe and so the system is secure.

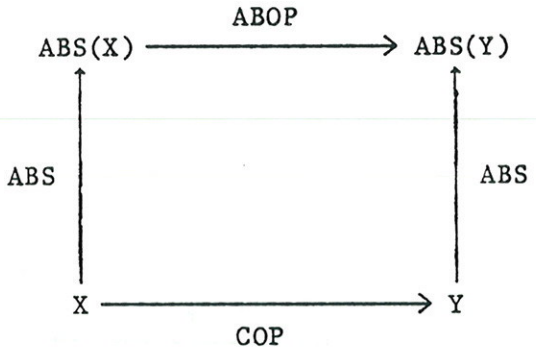
But does this procedure really deserve the title `information flow analysis´? It is not so much an analysis as a statement of the obvious. The structure of the partitioned system is such that it is really no more than a number of `private´ systems joined together - in fact it

could be produced mechanically from the specification of the private `abstract´ systems given in Figure 3. Indeed, this mechanical production is so straightforward that the systems of Figures 3 and 4 could be considered as no more than notational variants of each other. We hardly need, therefore, to `analyse´ Figure 4 in order to discover its manifest security.

A proponent of information flow analysis might argue that there is no need for the very obvious partitioning used in Figure 4 and that the technique could be used to verify the security of a system much closer to that of Figure 2. (In particular, the main memory could be left unpartitioned since it is already shared in space rather than time). The immediate riposte is then: "why bother"? Since information flow analysis cannot verify a system in which objects are shared dynamically, it can never verify a system that is close to Figure 2 in any significant sense. The crucially important feature of the system given in Figure 2 is that its general and mapping registers are shared in a secure manner. This element of secure sharing, which is the whole raison d´être of security kernels and their verification, cannot be modelled in any system that information flow analysis can verify: for that method to succeed, there must always be, at bottom, a partitioned system. The only room for manoeuvre is in the extent to which this partitioning is made manifest rather than obscure and, in either case, the real problem - that of proving the security of the system as it is actually implemented - remains unbroached. To accomplish that step we must prove the implementation is correct with respect to the specification embodied in the partitioned system.

As I said in Section 4, one danger with this two stage approach is that if its first stage (verifying the security of the partitioned system) is a substantial task (as it may be in the case of systems which enforce multilevel security), then it may come to appear as the most significant step in the whole enterprise. The correct implementation of the partitioned system by means of the unpartitioned, shared system may then seem to be no more than a question of careful coding - something that one ought to prove is done correctly but in which the likelihood of error is so small that the enormous cost of proof is unjustified in practice. Thus, the KSOS verification plan requires full verification of the security of its design specifications but only "illustrative" proofs of the correctness of its implementation [BERS79]. But, at least in the case of isolation (and I shall consider more general cases shortly), this emphasis on the first stage at the expense of the second is the precise opposite of what is appropriate. The real issue is the secure management of dynamically shared resources - which can only be verified using the techniques employed at the second stage.

So let us now look more carefully at this second stage. In principle, it poses the same problem as verifying the correctness of an implementation of an abstract data type with respect to its specification. As we saw in Section 5, the central requirement for an implementation to be correct is that the following diagram should commute:



Here, ABOP is the operation in the partitioned system that corresponds to COP in the implemented system and ABS is the abstraction function from the states of the implemented system to those of the partitioned one. The fact that we are dealing with a partitioned system, however, will allow us to split the abstraction function into separate components and to exploit this separation in order to reduce the complexity of the verification process. This is accomplished as follows. The very partitioning of the partitioned system means that its state space is the cross product of the state spaces of the individual, private systems embedded within it. Thus

$$S = S_1 \times S_2 \times \ \ldots \ \times S_n \times S_q$$

where S is the complete state space of the partitioned system, S_i ($1 \leq i \leq n$) is the state space spanned by the i-coloured variables and S_q is that spanned by the single variable AR (coloured SYSTEM). Then instead of one monolithic abstraction function ABS: $\bar{S} \rightarrow S$ (where $\bar{S}$ is the state space of the implementation) we can factor out a number of `local' abstraction functions:

$$ABS_i : \bar{S} \rightarrow S_i \qquad (1 \leq i \leq n) \text{ and}$$

$$ABS_q : \bar{S} \rightarrow S_q$$

where, for all s in $\bar{S}$,

$$ABS(s) = (ABS_1(s), ABS_2(s), \ \ldots \ , ABS_n(s), ABS_q(s)).$$

The verification of the security of the partitioned machine was achieved by showing that each of its operations (except SWAP) references only variables of the same colour as the currently active regime. Thus, if X and Y are two states of the partitioned system and ABOP is one of its operations such that

$$X \xrightarrow{\text{ABOP}} Y$$

where $X = (x_1, x_2, \ \ldots \ , x_r, \ \ldots \ , x_n, x_q)$

and $Y = (y_1, y_2, \ \ldots \ , y_r, \ \ldots \ , y_n, y_q)$

and r is the identity of the currently active regime, then it follows that

$$x_i = y_i \qquad (1 \leq i \leq n,\ i \neq r),$$

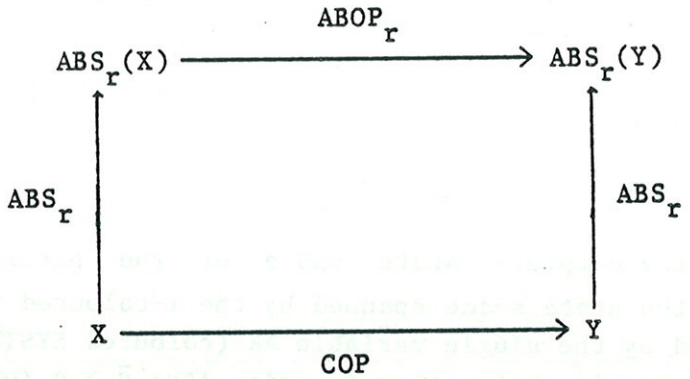
$$x_q = y_q$$

and

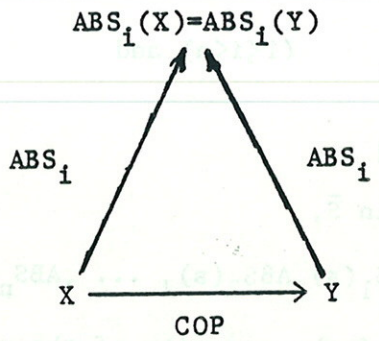
$$x_r \xrightarrow{ABOP_r} y_r$$

where $ABOP_r$ is a restriction of ABOP to the r-coloured variables.

Hence, the monolithic commutative diagram given earlier can be structured into the following two diagrams.



(where r is the active regime) and



(where i is an inactive regime).

The reader can observe the similarity between these diagrams and those which introduced proof of separability in Section 5. Although I have glossed over many details, it should be clear that, mutatis mutandis, the proof that a concrete, shared system is a correct implementation of a partitioned system is really no different, in principle, to its verification by `proof of separability´.

The similarity between these two methods is incomplete in one significant respect, however. Proof of separability requires more than just the commutativity of the diagrams above: it also imposes four additional constraints governing the behaviour of I/O devices and the

instruction sequencing mechanism. These constraints (which were listed on page 36) reflect issues of major importance to the security of computer systems; issues which are not addressed by, and cannot even be expressed within, the model underlying the two-stage `information flow analysis plus correctness´ method of verification. Robinson, one of those responsible for the verification of KSOS, puts it as follows [ROBI79]:

> "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."

It is not surprising that techniques based on information flow analysis should fail to address issues such as interrupt handling and asynchronous I/O devices - for they were not developed for the purpose of verifying security kernels. In her Thesis [DENN75, p9], D.E. Denning proposed information flow analysis as a method to:

> "certify the secure execution of a program in an otherwise secure system".

Similarly, Feiertag´s model was constructed to provide a basis for verifying the `Secure Object Manager´ (or `SOM´) of PSOS [FEIE77, NEUM77] - for which purpose it is perfectly suitable. This 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 interpreter - it behaves like a hardware extension and executes instructions on behalf of its user regimes. The identity of the regime on whose behalf it is operating at any one time is not indicated by a tag affixed to the instruction by some external agent, but is determined by the kernel´s own state. Thus the use of Feiertag´s model, unchanged, to justify the use of information flow analysis for the verification of the KSOS kernel [ANON78a] is of dubious validity and any attempt to verify a security kernel in that way must be incomplete in the areas I have mentioned.

It follows that the credibility of verifications based on information flow analysis is seriously undermined - for some of the trickiest code in a security kernel is concerned with exactly the issues ignored by this method of security verification, notably the management of asynchronous I/O and interrupt handling. Penetration audits have shown that these are typically the weakest points in the security of traditional systems and just the ones, therefore, that really need the rigorous scrutiny that verification provides. Unlike those for information flow, the model which underlies proof of separability [RUSH81b] explicitly addresses the interpretive character of a security kernel and thereby provides a sound and complete method for verifying the security of such systems. (Since no illustration or justification of that claim is

---

* by which he means those of information flow analysis

provided here, the reader may justifiably regard it with scepticism. I hope to allay such doubts in a subsequent report.)

Not only does proof of separability address issues outside the scope of techniques based on information flow analysis, but there is also a considerable difference in the mental attitudes engendered by the two approaches. The `information flow analysis plus correctness verification' technique encourages us to view the system from outside, to see it as a single, monolithic entity which requires analysis in order to discover the isolated regimes hidden within it. Proof of separability, inverts this perspective and invites us to examine the system from the viewpoint of the individual regime. The complete system is then seen as a fairly straightforward synthesis of these individual regimes.

One of the benefits (some might say, the principal benefit) that accrues from the attempt to verify a system lies in its impact on the design of that system. The requirements of proof impose a discipline and a frame of mind that encourages good design: complex, muddled designs do not yield to simple proofs of their correctness. Not only does verification enable us to prove that our design is right, but it helps us to get it right in the first place. It follows that a verification technique may be judged not only on its mathematical soundness, but also on its ability to identify the really crucial design issues and to provide the insight which leads to their mastery. I suggest that security verification by information flow analysis followed by correctness verification, fails to provide this insight and tends to cloud, rather than clarify, the central issues.

In contrast, I claim that proof of separability not only provides a sound technique for verifying the security of those kernelised systems that enforce a policy of isolation, but also lays bare the central features of their design and encourages a systematic approach to that design.

This conclusion might seem to be a consequence of the fact that I have concentrated, so far, on the very special security policy of isolation: proof of separability is particular to that policy and must surely fail when confronted by kernels which enforce the more general policy of multilevel security. In the next section, I shall argue that this is not so - or, rather, I shall argue that multilevel secure kernels are a chimera and that to enforce multilevel security within a security kernel is a fundamental design error. The correct role for a security kernel within a multilevel secure system is exactly that which underlies proof of separability.

7. MORE COMPLEX SYSTEMS AND POLICIES

So far we have only considered the security policy of isolation - a rather special policy which probably appears to be of rather limited practical interest. In this penultimate section I want to examine the additional problems that arise when designing and verifying systems to enforce more complex security policies. I shall argue that to embed the policy in the kernel, as is the current practice, leads to considerable difficulties and that it is better, as well as more natural, to distribute responsibility for policy enforcement to the individual components of the system. The only responsibility of the kernel is then to severely limit the extent to which different components can communicate or interfere with each other - a task which is very similar to imposing a policy of isolation upon them. Thus, isolation may be regarded, not merely as a limited and special case of the more sophisticated policies, but as the fundamental basis for their enforcement.

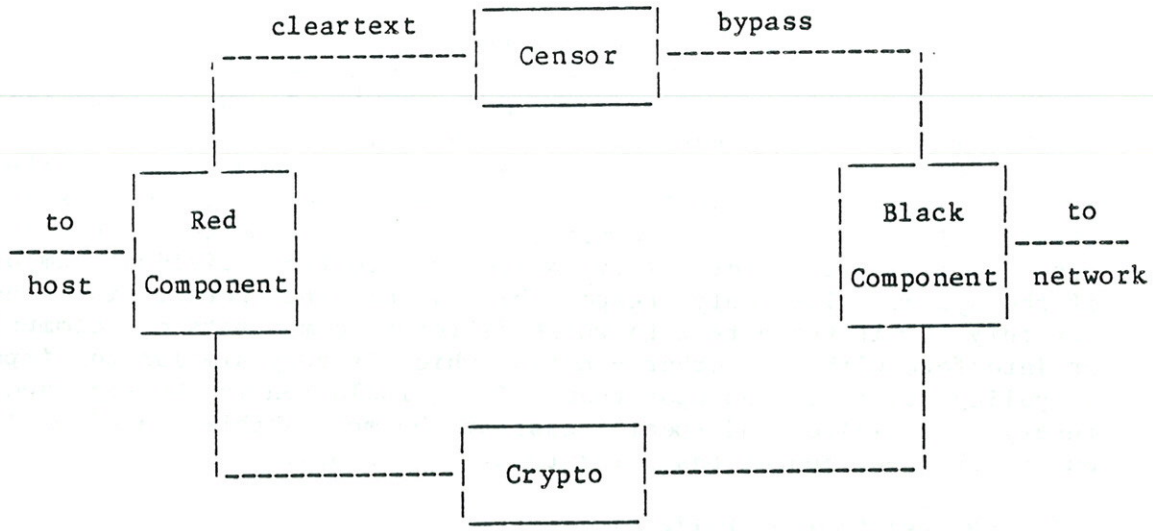
7.1. Channel Control Policies

Before moving on to consider multilevel policies, I want to introduce a quite different, but crucially important type of policy which can be illustrated very naturally in connection with a design for a `secure network front end´ (an `SNFE´) - that is a device which is interposed between a network and its host machines in order to provide end-to-end encryption.

There is more to the provision of secure communication through a network than simply inserting a cryptographic device (a **crypto**) between each host machine and the network. Some of the design issues have been discussed by Auerbach [AUER80] and a particular design is described by Barnes [BARN80]. Basically, the issues are as follows. As well as a crypto, the SNFE must certainly contain components for handling the protocols and message buffering 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. 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 cooperation (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 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.

Plainly, the role of the censor is critical to this design - but that is not my present concern. A perfect censor would be useless if any additional, uncensored, communication path directly connected the



red and black components. The important issue here then is not whether red and black can communicate, but what channels are available for that communication. The channel via the censor is permitted; we must prove that there are no others. This is an example of what I call a **channel control policy** since it is concerned with the identity of the channels through which information may flow, not simply with the presence or absence of flow.

We now have to consider how to verify a kernel which enforces a channel control policy. Techniques based on conventional information flow analysis are inapplicable, not only because they fail to identify the channels through which information may flow, but also because they assume a transitive flow relation. In the example, this would lead to the conclusion that, since information is allowed to flow from red to censor and from censor to black, direct flows from red to black must also be allowed! These and other difficulties encountered when trying to verify what are basically channel control issues in an environment based on information flow analysis are described by Ames and Keeton-Williams [AMES80].

I propose that, as in the case of isolation, the natural way to verify channel control policies is to relate the behaviour of the system actually available to that of an idealized system whose own security is obvious. In the present case, the ideal realisation of the system is surely the one sketched above - in which each of its four components is allocated to a private, physically separate machine and the communications channels between them are provided by external wires. In this ideal case, the absence of any communications channels other than those permitted by the policy can be established by direct physical examination.

In this realization of the system, the only software which performs a security-critical task is in the censor (the crypto is a trusted physical device); security is otherwise achieved by the physical distribution of the components and the physically limited communication paths provided between them.

In the real world, however, we have to construct and verify a concrete implementation of this ideal in which the red, black, and censor components, and the communications channels between them are all internal to a single, shared system. In order to keep the implementation close to its idealized model, it seems natural to base it upon a security-kernel which, as far as is possible, isolates each component within a separate regime. But how are we then to control the communication paths between the components when these are not physical wires but properties (and presumably subtle ones) of the kernel software? The idealised model gives us a clue: if we cut the communication paths that are allowed, then in the absence of illicit paths, the subsystems become isolated. So if we could `cut´ the allowed communication paths provided in the concrete system, and then verify that the resulting system satisfied a policy of isolation, we should have achieved our objective of proving the absence of illicit communication paths. The previous sections have shown us how to verify the policy of isolation so all that remains is to discover how to `cut´ a communication path that is not a tangible piece of wire but a property of some software.

The solution to this problem is easily found once we consider how communication is actually achieved in software - that is, by the use of shared variables. If one regime can communicate with another, then there must, at bottom, be some variable (or, in general, a set of variables) which the sender can write and the receiver can read. Suppose, then, that two regimes, A and B, are allowed to communicate through a shared variable X. 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, surely, 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.

An example should make this technique clear. Suppose the system of Figure 2 were to be augmented by the addition of a facility to allow regime 0 to write to regime 1. We could achieve this by adding a state variable X to the system together with two new operations READ and WRITE specified as:

```
OPERATION READ
PRECONDITIONS
    AR = 1
EFFECTS
    R(0)´ = X
END
```

```
OPERATION WRITE
PRECONDITIONS
    AR = 0
EFFECTS
    X´ = R(0)
END
```

Now to verify that the channel provided by the variable X is the only one in the system, we simply replace the appearance of X in READ by X1 and that in WRITE by X2 and verify (using the techniques of the previous sections) that the resulting system enforces a policy of isolation. (Notice that the selection of which instances of X to replace by X1 and which by X2 is a matter of judgement; choose `wrongly´ (in this example, replace both instances by X1, say) and it will not be possible to establish isolation for the resulting system - but this doesn´t affect the soundness of the method, only its success.)

This is an indirect argument and may appear specious: 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.

Once we have established that no inter-regime communication is possible, save over the channel provided by X, it remains to work out which regimes have access to this communication channel, and in which directions information can flow over it. Now the only way to transmit on the `X channel´ is by invoking an operation which can change the value of X, and the only way to receive is through an operation which can read this value. In the case of the example above, inspection of the **PRECONDITIONS** to each operation reveals that only regime 0 can transmit (through WRITE) and only regime 1 can receive (through READ) - and so we have verified the desired policy. Naturally, more complex arguments might be needed in the case of systems less trivial than this example, but the same underlying principle will remain.

In the case of the SNFE design sketched earlier, it is clear that there must be no direct communication channels connecting the red and black regimes: all traffic must pass through either the crypto or the censor. The absence of illicit communications channels is not the whole story, however: it will often be necessary to impose constraints on the behaviour of those channels which are allowed.

Suppose, for example, that the operation of the censor is to perform checks on `tokens´ submitted by the red regime and then, if they are satisfactory, to copy them over to the black regime. Suppose also that the censor does not keep its own copy of the token it is checking, but uses the one stored on its communication channel with the red regime. Then there is a clear danger of a security flaw if the red regime can change the value of the token stored on this channel between the time when the censor reads it for checking purposes and the time when it reads it again for copying to black. To avoid this danger, we must either provide a censor which protects itself against such attacks (for example, by reading the value of each input token only once), or else the channel over which the red regime submits its tokens must have properties which prevent such attacks ever being mounted.

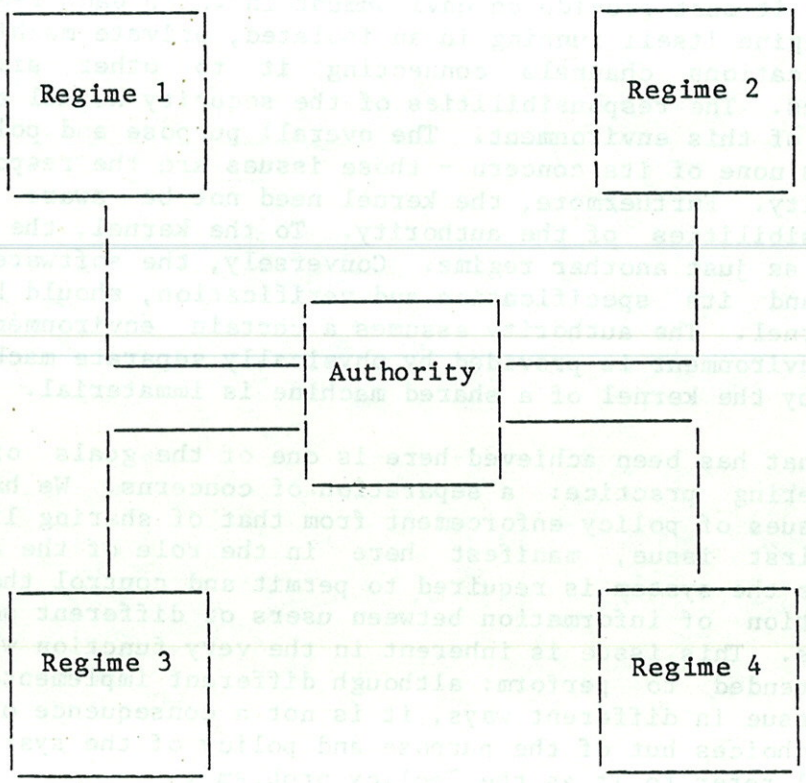
If the censor really were housed in its own physically separate machine, then we should probably have to adopt the first course of action: a censor so exposed must be prepared to counter arbitrary attacks upon its I/O interfaces. Alternatively, each separate machine in the system could contain a trusted subsystem which is guaranteed to impose an acceptable discipline on communications between machines. Of course, such a subsystem would need to be verified and would have much of the character of a security kernel. This suggests that the use of separate machines for each component of a secure system might not be quite the `ideal´ alternative to the use of a shared machine as it at first appears.

If the censor is housed in one of the regimes supported by a shared machine, then the security kernel of that machine can provide the censor

with a benign communications environment. In particular, since it provides all the inter-regime communications channels, the kernel can impose a suitable discipline on the use of those channels. In the case of the channel between the red regime and the censor, a suitable discipline is that provided by the abstraction of a bounded buffer [HOAR74], where only red can add tokens to the buffer, and only the censor can remove them. Proofs of the properties required of the censor may then assume this discipline is enforced at its I/O interfaces, while the kernel verification must be extended to prove that this discipline is enforced. The construction of suitable proofs for these properties is beyond the scope of this introductory survey but will be discussed in a subsequent report.

7.2. Multilevel Policies

The security of the design for the SNFE outlined above depends partly on the task performed by the censor and partly on the channel control policy enforced by the kernel. A similar division of responsibility can be adopted in the case of multilevel secure systems. Consider, for example, the very simple system sketched below.



The idea here is that four isolated single-user regimes are to be allowed to communicate with one another via a central `authority´ which acts as a `forwarding agent´. If the user in regime 1 wishes to send a message to the user in regime 3, then he must send it via the authority which has the power to decide whether or not to forward the message to

its intended recipient. The decision may depend on the content of the message, or on the security classifications of the users currently associated with regimes 1 and 3. As a particular form of the latter case, the authority can impose a multilevel policy - only forwarding messages if the classification of the receiver is at or above that of the sender.*

Obviously, the authority is crucial to the security of this system and must be proven to enforce its stated policy. The important point, however, is that just like the censor in the previous example, the role of the authority can be fully understood and specified, and its implementation proved correct, at a level of abstraction in which the authority is considered to be physically isolated from the user regimes, and the user regimes from each other: we may imagine that each regime runs in its own private machine. There is, at this stage, no need for a security kernel: the software in the authority regime is an ordinary program that provides a fixed single service - that of receiving and forwarding messages; it supports no user programming.

The need for a security kernel only arises when the physically isolated machines and the external communications channels of this idealized system are replaced by an implementation in which the four user regimes, as well as the authority and its communications channels, all share a single machine. In this case, a security kernel is needed to preserve the assumptions under which the idealized system was proved secure: it must provide an environment in which each regime can continue to imagine itself running in an isolated, private machine with external communications channels connecting it to other similarly isolated machines. The responsibilities of the security kernel end with the provision of this environment. The overall purpose and policy of the system is none of its concern - those issues are the responsibility of the authority. Furthermore, the kernel need not be aware of the special responsibilities of the authority. To the kernel, the authority should appear as just another regime. Conversely, the software of the authority, and its specification and verification, should be independent of the kernel. The authority assumes a certain environment, but whether that environment is provided by physically separate machines or is simulated by the kernel of a shared machine is immaterial.

What has been achieved here is one of the goals of good software engineering practice: a separation of concerns. We have distinguished the issues of policy enforcement from that of sharing limited resources. The first issue, manifest here in the role of the authority, arises because the system is required to permit and control the sharing or communication of information between users of different security classifications. This issue is inherent in the very function which the system is intended to perform: although different implementations may tackle this issue in different ways, it is not a consequence of any implementation choices but of the purpose and policy of the system. Accordingly, I shall refer to it as the `policy problem´.

---

* Of course, for this to be possible, there must be some additional mechanism (an **authentication server**), not shown in the diagram, to inform the authority of the identity and security classification of the user currently associated with each regime. I shall return to this topic later.

The second issue, which I call the `sharing problem´, arises only in those implementations where different users or system components share the same hardware resources. This issue is not a consequence of what the system is required to do, but of how it does it. The security kernel of our example system is a mechanism for solving the sharing problem: it makes it possible for conceptually separate regimes to share the same machine in a secure manner.

The most important benefit of this separation of concerns is that it enables us to handle the problem of `trusted processes´. This will become apparent later when I discuss systems more complex and realistic than the elementary example considered so far. A related benefit can be seen even in this simple example, however. Because the policy and the sharing problems are handled separately, by the authority and the kernel respectively, we can select specification and verification techniques appropriate to each problem independently of the other.

The purpose of the kernel is to enforce a channel control policy on five isolated regimes, one of which has communications channels to each of the other four. This can be verified by the method of `proof of separability´, augmented as described earlier to handle the control of communications channels. The function of the authority, on the other hand, is to switch a number of multilevel data streams. This role can be described by the SRI model of multilevel security [FEIE77] and from this it follows that information flow analysis can be used for its verification.

This proposal may surprise readers who formed the impression that Sections 4 and 6 of this paper condemned information flow analysis out of hand as a security verification technique. The explanation for this apparent inconsistency is that the differences between a kernel and an authority are sufficiently significant that the objections to information flow analysis as a kernel verification technique do not extend to its application to an authority.

Recall that the first objection to information flow analysis as a kernel verification technique is that it fails to address matters relating to interrupts and the asynchronous operation of I/O devices. However, the reason why these issues loom so large in kernel verification is that one of the purposes of a kernel is to shield its regimes from exactly these concerns: the kernel designer must worry about interrupts and malicious I/O devices just so that those who write application software need not. Operating, as it does, within the protected environment created by a kernel, authority software may be considered as a sequential program with communications to the outside world through well-behaved I/O channels. Thus the first objection to information flow analysis does not apply to its application to non-kernel software.

The second problem with information flow analysis is that it cannot cope with objects whose security classifications change during execution. At the lowest level, basic machine resources such as general registers have to be used for many different purposes and their classification must change with their use. As a result, and as we have already seen, information flow analysis cannot be applied to implementation-level kernel descriptions, but only to high-level `design specifications´. The task of proving that an implementation retains the security properties of its specification is left to conventional correctness

verification. This correctness verification step is likely to be far more complex and costly than the information flow analysis step, but without it the verification exercise has little point.

It might seem that this argument should apply in the case of an authority just as in the case of a security kernel. Strictly speaking, of course, this must be true. Information flow analysis can, indeed, provide only part of the evidence needed to verify the security of an authority - for if information flow analysis is not applied to the actual implementation of the authority, but only to its specification, then it cannot attest to the security of that implementation. However, there is a difference between an authority and a kernel which lessens the force of this argument and allows us to conclude that an authority, though not a kernel, can be adequately verified by application of information flow analysis alone. The difference concerns the extent to which the elaboration from verified specifications to executable code can be performed automatically, by a compiler.

If a specification of an authority has been proved secure by information flow analysis, then any implementation of that specification which preserves the same input/output behaviour will be acceptable and will also be secure. If a compiler is available for the specification or programming language concerned, then this elaboration from secure specification to secure implementation can be left to the compiler. Strictly speaking, of course, we should verify the correctness of the compiler but in practice we can surely trust it without proof - for the possibility that a compiler which has performed correctly many thousands of times should this time deliver code that also behaves apparently correctly (for we will certainly test it), but which contains a security flaw that can be exploited through the very limited input/output interface provided by the authority is surely negligible.* To fear security threats from this quarter is otiose if not paranoid. The resources needed to verify their absence would be better employed elsewhere.

Now whereas the authority is an `ordinary' program, whose input/output behaviour is completely defined in its specifications, a security kernel is best understood as an interpreter for other programs. Or, more accurately, it is best understood as part of an interpreter - for much of the interpretation of user programs is performed directly by the hardware. In consequence, certain hardware resources (such as the general registers) are directly visible to user programs and this constrains the way in which kernel specifications can be elaborated into code. It is necessary, for example, that both the UNCLASSIFIED and the TOP SECRET sets of general registers should, at different times, be mapped onto the single set of hardware registers. Of course, the variables of the authority software must also be mapped onto hardware registers but the crucial difference is that the users of the authority are not allowed to manipulate those hardware registers directly.

It is the basic task of a security kernel to manage machine registers `securely' - yet at the `design specification' level, these registers are not present in any realistic form. In truth, `design specifications' for a kernel are hardly specifications at all, for they do not

---

* I am assuming a stable, much used compiler, not specially written or modified for the task in hand.

completely define its input/output behaviour: the input/output interface of a kernel includes all the machine registers that are visible to user programs and to attached I/O devices. Thus, unlike those of an authority, high-level kernel specifications cannot be elaborated into code automatically: human ingenuity is required to reconcile the visible but fixed resources and interpretations provided by the hardware with those assumed by the specifications. It follows that an additional verification step is needed in the case of a kernel that is not required in the case of an authority.

Let me summarise the discussion so far. We have distinguished the `policy problem´ and the `sharing problem´ as separate issues within the overall problem of secure system design and implementation. This distinction leads to a system design in which all policy issues are the responsibility of an `authority´ while the sharing problem is handled by a security kernel. The separation of concerns so achieved is aesthetically pleasing and of practical benefit since it enables the kernel and the authority to be verified separately, and by different techniques; in each case we may employ the most appropriate method: proof of separability for the kernel and information flow analysis for the authority.

Now it might seem that this clean and simple design will become less so when we add the other functions that must be provided by a realistic system. How are we to fit a secure line-printer spooler, file store and network interface into this neat dichotomy of authority and kernel? The answer is that we do not fit these new services into either of the existing components but add them as further self-contained `servers´. Just as the authority is conceived as an autonomous, isolated component dedicated to the provision of a multilevel secure message forwarding service, so multilevel secure file and print servers can be added as further autonomous and isolated components provided with limited connections to each other and to user regimes through dedicated communications channels. Naturally, since each of these new components performs a security-critical task, their design and implementation must be shown to maintain the security of the overall system.

The security properties required of a multilevel secure file server, like those required of the message-forwarding authority, can be described by the SRI model and can be verified by information flow analysis. A secure print server, on the other hand, must satisfy rather different requirements. For example, the `Line-Printer Daemon´ (despooler) of KSOS is required to:

> "... apply the appropriate headers, footers, and burst page information to correctly mark the classification of the output. Markings shall be in accordance with DoD Directive 5200.1-R. The Daemon shall create auditing records noting the creation of classified output." [ANON78b, Section 3.2.2.2.3]

These requirements are additional to those of multilevel secure information flow and cannot be verified by information flow analysis. Further-

more, a print server may not merely require to add to the normal multilevel constraints, but also to break them. The KSOS Line Printer Daemon, for example:

> "shall be privileged to violate the security *-property, in order to allow it to delete files after printing." [ANON78b, Section 3.2.2.2.1]*

Other specialised system components, such as those responsible for authenticating users as they log in and with initialising the system at start-up time, although they are clearly vital to security, seem to have requirements even further outside the normal rules of multilevel secure operation.

What we are seeing here is that just as the overall security problem may be divided into policy and sharing issues so, within the policy problem itself, we may distinguish a number of sub-problems, each specific to the precise role and function of an individual component. Although a file server and a print server may both be parts of a system designed to enforce one multilevel security policy, the policies that govern their own behaviour cannot simply be that overall policy in microcosm but must be particular to their individual function and to their role within the larger system. The properties required of a secure print server, for example, depend as much on the fact that it is a print server as on the security policy that is to be enforced.

It follows that the overall security of a complex system cannot be founded on the centralised enforcement of a single, all-embracing security policy - yet this is precisely the approach used in existing systems such as KVM/370 and KSOS. In these systems, the intention is that all security-critical software should reside in the kernel and that the kernel should be verified to enforce multilevel security. From the immediately preceding discussion it is apparent that this approach is incompatible with the provision of certain necessary system functions and it is hardly surprising to learn that both KSOS and KVM/370 contain security critical `trusted processes´ that reside outside the kernel and which are allowed to flout the security controls enforced by the kernel. These `semi-trusted´ processes, as they are called in KVM/370 or `privileged NKSS´ (Non Kernel System Software) in KSOS, perform all the

---

* If the print server and all 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 whether their jobs have been printed. For this reason, it is usual for spool files to be classified at the level of their owners - while the print server continues to run at the highest classification level so that it can read spool files of all classifications. But then the print server cannot delete spool files after their contents have been printed, nor can it even inform their owners of this fact - for such actions conflict with the *-property of multilevel security. So, in order to provide an acceptable user interface, while avoiding a proliferation of used spool files, it seems necessary that a print server should be allowed to violate the *-property.

system functions which don't seem to fit in with the general security rules of the system. In KSOS, for example, the privileged NKSS contains:

> "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.)." [BERS79, p365]

while in KVM/370, the semi-trusted processes:

> "control access to global system resources (page scheduling, direct access storage, slot allocation). They have the potential to leak data indirectly." [SCHA77, p405]

The contract for KSOS required proof of the security of all the privileged NKSS, but I understand that this has not, and will not, be achieved. In KVM/370, the semi-trusted processes were "audited" (i.e. checked informally) rather than verified [GOLD79]. The existence, and apparent difficulty of proof, of trusted processes do not imply that the KSOS and KVM/370 kernels are ill-designed. Rather, to quote from Berson and Barksdale again:

> "To a large extent, they represent a mismatch between the idealizations of the multilevel security model and the practical needs of a real user environment." [BERS79, p365]

The basic problem is not the individual design of the KSOS and KVM/370 kernels; it is the whole conception that a security kernel should act as a centralized agent for the enforcement of a uniform system-wide security policy that is fundamentally flawed. The tasks performed by trusted processes just cannot be accomplished under the restrictions that normally govern multilevel secure operation and so special privileges are required in order to evade those restrictions. Responsibility for the security of the total system is therefore divided between the kernel and the trusted processes. But this division is not a clean one: it does not represent a separation of concerns but a confusion of the same. The kernel has to provide the trusted processes with the means to evade its own control, yet the trusted processes are not autonomous - they rely on residual protection afforded by the kernel. Thus the security properties of the kernel are inextricably bound up with those of the trusted processes. Neither kernel nor trusted processes can be fully understood independently of each other, while in conjunction they can hardly be understood at all!

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 little real justification for speaking of the `verification' of the security of such systems at all. Landwehr, for example, observes:

> "... in the final version of their model, Bell and La Padula did include trusted processes. What is not included in their exposition is a technique for establishing when a process may be trusted." [LAND80, p46]

Instead of attempting to construct secure systems around a centralised mechanism that imposes a single security policy over the entire system, it is surely more natural seek a system structure which allows each component to make its own contribution to the security of the overall system and which treats all contributions equally - as befits the `weakest link´ nature of security. We should not elevate the security requirements particular to one component, or class of components, to a special status and attempt to impose them system-wide at whatever inconvenience to components with different requirements.

I suggest, therefore, that it is helpful to conceive of secure systems as distributed systems composed of communicating but otherwise isolated components, each dedicated to a single function within the overall system. Some components will service the users of the system and will support untrusted user-written applications programs. Other components will be trusted to provide services such as secure communication between users and secure access to shared data. The main 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 print server, for example, requires special services of the file server (i.e. the ability to delete spool files of all security classifications) 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 trusted 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 print 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 de-spooler of a kernelized system a dispensation to flout the *-property.

Under the `distributed´ approach to secure systems design, the presence of a security kernel is a totally independent issue: one that only arises when a conceptually distributed system is actually implemented within a single, shared machine. Thus the `trusted components´ of a distributed system are not merely `trusted processes´ under a different name, but may be thought of as trusted processes whose responsibilities have been so greatly magnified (at the expense of those of the kernel) that their role has undergone metamorphosis. Trusted components are autonomous (although they may cooperate with each other and share a common purpose) and can be understood independently of the kernel. They require no special privileges of the kernel, neither does the kernel need to know anything of their responsibilities. Policy enforcement is not the concern of a security kernel: its only purpose is to maintain the abstraction of an environment composed of distributed, independent components within an undistributed, shared implementation.

This approach decouples the verification of the different system components from one another and from the verification of the kernel. Once the role and security requirement of every component has been understood and specified in relation to the larger whole, each component may thereafter be studied and verified independently of the others. Some components may conform to the SRI model of multilevel security, in which case information flow analysis will provide a suitable technique for their verification. Other components may present new and quite different problems of security specification and verification. Selecting (or developing) appropriate techniques for the resolution of these problems may be a difficult and challenging task - but we are surely in a better position to tackle these difficulties when the function and responsibilities of each component are isolated and exposed, as in the distributed model, rather than hidden among the special privileges granted to `trusted processes´.

The approach proposed here is not new - although its presentation is intended to be more coherent and its foundation on `proof of separability´ is believed to be more soundly based than earlier proposals of a similar vein. The idea that a security kernel should simulate a `distributed´ environment within a single machine is very close to the notion of a `Virtual Machine Monitor´ (a `VMM´). The application of VMMs to security was proposed by Anderson in 1974 [ANDE74] and, in more detail, by Popek and Kline in the same year [POPE74b]. In a panel session at the AFIPS National Computer Conference of 1974, Popek articulated essentially the same approach as that advocated here:

> "Let the basic kernel of a general purpose operating system provide for supporting and isolating simple processes. The only primitive sharing facility included might be the ability to arrange shared read-write segments. Then, a file system, with its own kernel, could be built in one process, and whenever a user wished to act on a file, he would communicate with the file system through a shared segment. The original kernel´s task of process isolation and the responsibilities of the file system´s kernel have been separated, decreasing complexity." [POPE74a, p977]

Obviously this approach is not restricted to multilevel policies: its application to specialised systems with unusual or conflicting security requirements remains quite natural. As the next section will illustrate, implementations based on conventional kernels are not so flexible.

7.3. Special Purpose Policies and Systems

Due to modern hardware developments, large general-purpose multiuser mainframe-based systems no longer dominate data processing operations as they did only a few years ago. Systems based on a number of interconnected, relatively small machines, each dedicated to a single user or providing a single service, are becoming increasingly attractive.

With the arrival of such systems, the problem of constructing multilevel secure general-purpose operating system assumes less importance than formerly. In its place come a number of new problems, each specific to a single application: for example, the provision of secure

data-bases, file servers, and network front ends. A number of such applications have been described by Woodward [WOOD79] while network applications are also considered by Padlipsky et al. [PADL79].

It is not obvious that the systems and security models developed for the `old' problem of multilevel secure general-purpose operation will be appropriate, or even applicable, to the problems of specifying and enforcing the security properties required for these new applications. The secure network front end outlined in Section 6.1, for example, has security requirements that bear no relationship whatever to the multilevel models of Feiertag or Bell and La Padula - and to base its implementation on a conventional multilevel secure kernel would be to take several steps in the wrong direction. Regrettably, however, there is a tendency to assume that the multilevel model (and in particular the *-property) is fundamental to all notions of security and that kernelized multilevel secure systems provide the natural base for all applications that require the enforcement of some notion of security.

This attitude leads to multilevel security becoming a Procrustean Bed to which all applications must be ruthlessly accommodated. An interesting example is provided by the ACCAT Guard [WOOD79].

The Guard is 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 are displayed to a human `Security Watch Officer' (SWO) who must decide whether they may be declassified to the level of the LOW system. The security properties demanded of the Guard include, for example, the requirements that all messages from HIGH to LOW are displayed to the SWO in their entirety and that only the ones which he accepts are passed through to the LOW system. These requirements have nothing in common with the ss- and *-properties of conventional multilevel security and, furthermore, all information flow from HIGH to LOW is in direct contravention of the *-property. It must be considered extraordinary, therefore, that KSOS was chosen as the basis for the implementation of the Guard - for since the KSOS kernel permits information flow in only the LOW to HIGH direction, all HIGH to LOW transfers have to be accomplished by trusted processes whose basic purpose is to get round the fundamental security principle of the KSOS kernel! It is not clear how the use of the KSOS kernel has contributed to the overall security or verifiability of this design and it is certainly no surprise to learn from Landwehr [LAND80, p46] that:

> "Verification of the trusted processes to be used in the Guard has consumed far more resources than originally planned."

I have argued that the centralised enforcement of a single all-embracing security policy is inappropriate, even for a system whose overall purpose was in general accord with that policy. In the case of a system like the Guard where two incompatible policies have to be enforced simultaneously (one for LOW to HIGH transfers and another for HIGH to LOW) it is not merely inappropriate, it is grotesque.

The most obvious characteristic of the Guard is that it is fundamentally dichotomized: the management of LOW to HIGH transfers is quite different and logically separate from that of HIGH to LOW transfers. The only sensible way to proceed is to recognize this dichotomy and

structure the system as two distinct subsystems, corresponding to the two directions of transfer. A security kernel is then needed to allow the subsystems to co-exist in the same machine without interfering with one another. The enforcement of their individual security policies is the responsibility of the subsystems themselves: the kernel cannot assist one without hindering the other. Thus the problem of the Guard can be broken down into three completely independent sub-problems: the LOW to HIGH subsystem, the HIGH to LOW subsystem and the security kernel. In this way we obtain a separation of concerns and a corresponding simplification that is quite absent from the KSOS-based implementation.

Whether the benefits of this simplification would be significant in practice remains to be seen. The Guard performs a complex function and its verification must necessarily be rather difficult. It would be facile to claim that all the difficulties and complexity described in [AMES80] can be removed by the techniques advocated here. Nonetheless, it is, perhaps, significant that a successor to the ACCAT Guard (this is the `LSI Guard´ [CRAI80]) is not based on a KSOS-type kernel.

The proposals for the design and verification of secure systems which have been presented here are currently being evaluated in practice. An SNFE similar to that outlined in Section 7.1 has been built by T4 Division of the U.K. Royal Signals and Radar Establishment and has been in use for some time now as part of their `Pilot Packet Switched Network´ [BARN80, MAST80]. Although the design and implementation of this system preceded the formulation of the verification-oriented rationale presented here, it is entirely consonant with it. Current work is aimed at the verification of this system. If this is accomplished successfully, it will have the singular distinction of being the smallest operational system for which security properties are claimed to have been verified.

8. CONCLUSION

Imagine an office in which a number of workers generate and process information recorded in files which are fetched and stored on their behalf by a clerk. Because of the sensitivity of the information concerned, a security policy is imposed: certain rules govern which workers may deal with which files. In order to enforce these rules we can isolate each worker in a separate sealed room. Each room is provided with a desk, a chair, a private filing cabinet and so on, and also with a mailbox through which files and written requests for files may be exchanged with the filing clerk who inhabits a common outer room. The security of this operation depends on the trustworthiness and diligence of the filing clerk: he must be relied on to refuse requests for files that are not in accord with the security policy. Since the workers are isolated from one another in separate rooms, however, they have no opportunity to communicate with one another and need not be trusted.

Now suppose that financial constraints dictate that only one room can be heated to a level sufficient to support cerebration: unfortunate workers consigned to an unheated room enter a state of suspended animation: they survive but are quite unconscious. Even in these straitened circumstances we can still provide a secure and productive environment for our participants if we enlist the aid of another trusted employee - the caretaker. The task of this functionary is arrange, periodically, for the inhabitant of the heated room to exchange places with one of the other participants. The caretaker has to ensure that when each newly arrived worker wakes in the heated room, he finds his surroundings exactly as when he was last conscious of them (apart from the possible arrival of material through a mailbox). Thus, the caretaker must look after the furniture and private working material of each worker and, in the case of the filing clerk, provide facilities that are denied to others (i.e. access to the central filing cabinets and to the mailboxes of other workers).

It seems clear that this scheme, though more complex, need not be any less secure than the first. Furthermore, the filing clerk and the other workers need not be informed of the new arrangements: they should perceive no change in their environment and be unaware of their occasional lapses into unconsciousness.

The point of this, perhaps rather strained, analogy to convey (especially to those readers who may have chosen to begin their examination of this paper here, rather than at its beginning) an intuitive basis for understanding my assertion that the design and verification of secure systems poses not just one, but two problems. I call these the `policy´ and the `sharing´ problems and they are exemplified here in the roles of the filing clerk and the caretaker, respectively.

The policy problem is a consequence of the task performed by the system and the security constraints it is required to meet. It is, in short, a consequence of what the system has to do. The system just described has to provide a number of workers with access to a central filing system subject to certain constraints on that access. This is achieved by placing a trusted filing clerk in charge of the filing system and requiring him to mediate all accesses by the other workers. Thus the filing clerk is a mechanism to handle the policy problem.

The role of the caretaker, however, is quite different. He is not required as a consequence of the purpose of the system, but of its implementation; he is needed because circumstances dictate that everybody has to share the use of certain resources - the heated room in this case. While the caretaker is not responsible for the security policy of the system, he is responsible for its ability to enforce that policy: both the filing clerk and the caretaker must behave properly if the overall system is to be secure. Security founders if the clerk delivers a file to a worker who should not be allowed to see it, and it also founders if the caretaker ushers a new worker into the heated room without removing material belonging to its previous occupant.

The point to be stressed is that the roles of the filing clerk and of the caretaker are fundamentally different to one another and should be studied independently of each other. It would be a grave error of design to combine both roles into one all-purpose general servant.

I am convinced that the cleanest separation of policy and sharing issues is achieved if secure systems are conceived as distributed systems in which the subjects of the security policy and the agents of its enforcement all reside in conceptually private and separate subsystems linked by appropriate communications lines.

The system designer must study the role of the components in his distributed design in order to determine the properties required of each of them individually so that, in combination, they enforce the security policy of the overall system. The security requirements of components such as file-servers can often be expressed as restrictions on information flow as formalised in the models of SRI and Mitre. The technique of information flow analysis can verify the security of such components very economically (since it is only necessary to consider the security classifications of variables, not their values). Other components, such as those which interact with the outside world (for example, print servers), or which influence the behaviour of other components (for example, authentication servers) have requirements expressed in terms of the actual values of quantities which they output, not just their security classification (i.e. they must produce the right answer, not just an answer that depends only on inputs below a certain classification level). Information flow arguments are not sufficient to verify the security of these components; more powerful methods are required.

The requirements of a security kernel are different again. A kernel's task is to allow conceptually distributed system components to share a single machine in perfect safety. To do this it provides each component with an environment indistinguishable from an isolated machine dedicated to that component alone. Existing techniques are not well suited to the verification of this property and I have argued that a specialised technique called `proof of separability' is the correct tool for this job.

I claim that this approach to the design of secure systems both exposes and narrows the interfaces between different components. By reducing the interdependence between the security kernel and the other trusted components it brings about a separation of concerns which lessens the complexity of the system and enables the security kernel to be studied and verified more or less independently of the rest of the system. On the other hand, by making explicit the communications

channels that run between certain components, the approach makes all too clear the delicate web of mutual interdependence and trust on which overall security depends. Making complexity manifest is no bad thing - especially if it leads to a more profound appreciation of the consequences of demanding too many `facilities´ in a secure system.

Secure systems are among the first real-world systems to be built under contracts which require formal verification of certain aspects of their behaviour. As such, they pose both an opportunity and a threat to those who believe that verification and verification-oriented design offer the best hope of advancing computer programming from the status of a craft to that of an engineering discipline.

The opportunity is obvious: for the first time, sufficient resources are being made available to allow practical application of formal specification and verification techniques. Much will be learned thereby. The danger is that expectations will be allowed to exceed realistic prospects. Verification systems are still in the early stages of development and our understanding of many important issues is still incomplete. Yet it is becoming routine that proposals for military systems should include the requirement that their security be formally verified. There must be a danger that the verification of these massive systems will become a meretricious exercise - its purpose being to impress the customer with its size, detail and formalism even if it leaves him uncertain of precisely what has been proved.

Verification of the security (or any other property) of a system is achieved by demonstrating, in a suitably formal manner, that the behaviour of the system is `consistent´ with that of a model which is sufficiently simple that it may be seen, directly and intuitively, to possess the desired property. The notion of `consistency´ must also be sufficiently natural that it can be accepted without question. A verification exercise has achieved its goal only when every step of the argument, from model to system, is so utterly compelling that any suitably trained person will be convinced of its veracity. Repeatedly, experience has taught us that our ability to reason about complex systems is limited and fallible. To have any hope of success, we must, at least for the time being, curb our ambition and strive for simplicity. As Hoare observed in his Turing Award Lecture:

> "There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that no deficiencies are obvious." [HOAR81]

It is not the purpose of verification to lend a spurious respectability to systems that follow the latter course - although commercial pressures may be leading in that direction.

Verification provides a coherent framework for mastering some of the myriad details that must be attended to if a computer system is to perform satisfactorily. It is not a tool for making large, complex systems as easy to manage as small simple ones. Indeed, one of the principle benefits of verification lies in its capacity to reveal complexity and thereby encourage good design. What is needed now are convincing demonstrations of the practicality and utility of this approach applied to (relatively small, but useful) operational systems.

ACKNOWLEGEMENTS

I am grateful to all those who commented on earlier versions of this paper, and especially to Tom Anderson, Derek Barnes, Flaviu Cristian, and Tom Parker.

REFERENCES

ABRI80 ABRIAL, J.R., "The Specification Language Z: Syntax and "Semantics"," Internal Report, Programming Research Group, Oxford University. (April 1980).

AMES80 AMES, S.R. JR. and KEETON-WILLIAMS, J.G., "Demonstrating Security for Trusted Applications on a Security Kernel Base," Proceedings of the 1980 Symposium on Security and Privacy, Oakland, Calif., pp.145-156, IEEE Computer Society (April 1980).

ANDE72 ANDERSON, J.P., "Computer Security Technology Planning Study," ESD-TR-73-51 (October 1972). (Two volumes).

ANDE74 ANDERSON, J.P., "Systems Architecture for Security and Protection," pp. 49-50 in Approaches to Privacy and Security in Computer Systems, ed. C.R. Renninger, NBS Special Publication 404, GPO SD Catalog No. C13.10:404, Washington, D.C. (1974).

ANDR80 ANDREWS, G.R. and REITMAN, R.P., "An Axiomatic Approach to Information Flow in Programs," TOPLAS Vol. **2**(1), pp.56-75 (January 1980).

ATTA76 ATTANASIO, C.R., MARKSTEIN, P.W., and PHILLIPS, R.J., "Penetrating an Operating System: a Study of VM/370 Integrity," IBM Systems Journal Vol. **15**(1), pp.102-116 (1976).

AUER80 AUERBACH, K., Technical Correspondence on "Secure Personal Computing", CACM Vol. **23**(1), pp.36-37 (January 1980).

BARN80 BARNES, D., "Computer Security in the RSRE PPSN," Networks '80, pp.605-620, Online Conferences (1980).

BECK80 BECKER, J., "Who are the Computer Criminals?," New Scientist, pp.818-821 (13 March 1980).

BELL76 BELL, D.E. and LAPADULA, L.J., "Secure Computer System: Unified Exposition and Multics Interpretation," ESD-TR-75-306, Mitre Corporation, Bedford, Mass. (March 1976).

BERS79 BERSON, T.A. and BARKSDALE, G.L. JR., "KSOS - Development Methodology for a Secure Operating System," AFIPS Conference Proceedings Vol. **48**, pp.365-371 (1979). National Computer Conference.

BEST80 BEST, E., "Information Flow in Nets," Proceedings of First European Workshop on Application and Theory of Petri nets, Strasbourg, France (September 1980).

BIBA77 BIBA, K.J., "Integrity Considerations for Secure Computer Systems," MTR-3153 Rev. 1, Mitre Corp., Bedford, Mass. (April 1977).

BURS81 BURSTALL, R.M. and GOGUEN, J.A., "An Informal Introduction to CLEAR, a Specification Language," in The Correctness Problem in Computer Science, ed. R.S. Boyer and J.S. Moore, Academic Press (1981).

COHE78 COHEN, E.S., "Information Transmission in Sequential Programs," pp. 297-336 in Foundations of Secure Computation, ed. R.A. DeMillo et al. , Academic Press (1978).

CRAI80 CRAIGEN, D. and BONYUN, D., "Two Projects in Program Verification," ACM Software Engineering Notes Vol. **5**(3), pp.12-13 (July 1980).

DELA79 DELASHMUTT, L.F. JR., "Steps Toward a Provably Secure Operating System," Proceedings of IEEE Compcon, pp.40-43 (Spring 1979).

DENN75 DENNING, D.E., "Secure Information Flow in Computer Systems," Ph.D. Thesis, Purdue University (May 1975).

DENN76 DENNING, D.E., "A Lattice Model of Secure Information Flow," CACM Vol. **19**(5), pp.236-243 (May 1976).

DENN77 DENNING, D.E. and DENNING, P.J., "Certification of Programs for Secure Information Flow," CACM Vol. **20**(7), pp.504-513 (July 1977).

DENN79 DENNING, D.E. and DENNING, P.J., "Data Security," Computing Surveys Vol. **11**(3), pp.227-249 (September 1979).

DENN80 DENNING, D.E., "Embellishments to a Note on Information Flow into Arrays," Software Engineering Notes Vol. **5**(2), pp.15-16 (April 1980).

ENDE72 ENDERTON, H.B., A Mathematical Introduction to Logic, Academic Press, New York (1972).

FEIE77 FEIERTAG, R.J., LEVITT, K.N., and ROBINSON, L., "Proving Multilevel Security of a System Design," Proceedings 6th ACM Symposium on Operating System Principles, pp.57-65 (1977).

GOGU78 GOGUEN, J.A., THATCHER, J.W., and WAGNER, E.G., "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," pp. 80-144 in Current Trends in Programming Methodology, IV: Data Structuring, ed. R.T. Yeh, Prentice-Hall, New Jersey (1978).

GOLD77 GOLD, B.D. et al., "VM/370 Security Retrofit Program," Proceedings ACM '77, pp.411-417 (1977).

GOLD79 GOLD, B.D. et al., "A Security Retrofit of VM/370," AFIPS Conference Proceedings Vol. **48**, pp.335-344 (1979). National Computer Conference.

GOSS80 GOSSER, M., MILLEN, J.K., and WILSON, W.F., "A Note on Information Flow into Arrays," Software Engineering Notes Vol. **5**(1), pp.28-29 (January 1980).

HEBB80 HEBBARD, B. et al., "A Penetration Analysis of the Michigan Terminal System," ACM Operating Systems Review Vol. **14**(1), pp.7-20 (January 1980).

HOAR72 HOARE, C.A.R., "Proof of Correctness of Data Representations," Acta Informatica Vol. **1**, pp.271-281 (1972).

HOAR74 HOARE, C.A.R., "Monitors: an Operating System Structuring Concept," CACM Vol. **18**(12), pp.549-557 (October 1974). (Corrigendum CACM 18(2) p. 95 (February 1975)).

HOAR81 HOARE, C.A.R., "The Emperor's Old Clothes (1980 ACM Turing Award Lecture)," CACM Vol. **24**(2), pp.75-83 (February 1981).

HSIA79 HSIAO, D.K., KERR, D.S., and MADNICK, S.E., Computer Security, Academic Press (ACM Monograph Series) (1979).

HUET80a HUET, G., "Confluent Reductions : Abstract Properties and Applications to Term Rewriting Systems," JACM Vol. **27**(4), pp.797-812 (October 1980).

HUET80b HUET, G. and OPPEN, D., "Equations and Rewrite Rules: a Survey," in Formal Languages: Perspectives and Open Problems, ed. R. Book, Academic Press (1980).

HUET80c HUET, G. and HULLOT, J-M., "Proofs by Induction in Equational Theories with Constructors," Proceedings 21'st Symposium on Foundations of Computer Science, pp.96-107, IEEE Computer Society (1980).

KNUT70 KNUTH, D.E. and BENDIX, P.B., "Simple Word Problems in Universal Algebras," pp. 236-297 in Computational Problems in Abstract Algebra, ed. J. Leech, Pergamon Press (1970).

LAMP71 LAMPSON, B.W., "Protection," Proceedings, Fifth Annual Princeton Conference on Information Sciences and Systems, pp.437-443 (1971). (Reprinted in ACM Operating Systems Review Vol. 8(1) 1974).

LAMP73 LAMPSON, B.W., "A Note on the Confinement Problem," CACM Vol. **16**(10), pp.613-615 (October 1973).

LAND80 LANDWEHR, C., "Assertions for Verification of Multilevel Secure Military Message Systems," ACM Software Engineering Notes Vol. **5**(3), pp.46-47 (July 1980).

LIND75 LINDE, R.R., "Operating System Penetration," AFIPS Conference Proceedings Vol. **44**, pp.361-368 (1975). National Computer Conference.

LIND76 LINDEN, T.A., "Operating System Structures to Support Security and Reliable Software," Computing Surveys Vol. **8**(4), pp.409-445 (December 1976).

LIPN75 LIPNER, S.B., "A Comment on the Confinement Problem," Proceedings of the Fifth Symposium on Operating Systems Principles, pp.192-196 (1975).

LOCA80 LOCASSO, R., SCHEID, J., SCHORRE, V., and EGGERT, P., "The INA JO Specification Language Reference Manual," TM-(L)-6021/001/00, System Development Corporation, Santa Monica, Calif. (June 1980).

MAST80 MASTERMAN, P.H., "The RSRE Pilot Packet-Switched Network," Networks '80, pp.277-292, Online Conferences (1980).

MILL76 MILLEN, J.K., "Security Kernel Validation in Practice," CACM Vol. **19**(5), pp.243-250 (May 1976).

MILL78 MILLEN, J.K., "Example of a Formal Flow Violation," COMPSAC '78, pp.204-208, IEEE Computer Society (1978).

MILL79 MILLEN, J.K., "Operating System Security Verification," M79-223, Mitre Corporation, Bedford, Mass. (September 1979).

MILN71 MILNER, R., "An Algebraic Definition of Simulation between Programs," 2nd International Joint Conference on Artificial Intelligence, pp.481-489, London (1971).

MUSS80a MUSSER, D.R., "Abstract Data Type Specification in the AFFIRM System," IEEE Trans. on Software Engineering Vol. **SE-6**(1), pp.24-32 (January 1980).

MUSS80b MUSSER, D.R., "On Proving Inductive Properties of Abstract Data Types," Proceedings 7'th Symposium on Principles of Programming Languages, pp.154-162 (January 1980).

McCA79a McCAULEY, E.J. and DRONGOWSKI, P.J., "KSOS - The Design of a Secure Operating System," AFIPS Conference Proceedings Vol. **48**, pp.345-353 (1979). National Computer Conference.

McCA79b McCAULEY, E.J., "KSOS: A Secure Operating System," Proceedings Compcon, pp.35-39, IEEE Computer Society (Spring 1979).

NAKA78 NAKAJIMA, R., HONDA, M., and NAKAHARA, H., "Describing and Verifying Programs with Abstract Data types," pp. 527-556 in Formal Description of Programming Concepts, ed. E.J. Neuhold, North-Holland (1978).

NAKA80 NAKAJIMA, R., HONDA, M., and NAKAHARA, H., "Hierachical Program Specification and Verification - a Many-Sorted Logical Approach," Acta Informatica Vol. **14**, pp.135-155 (1980).

NEUM77 NEUMANN, P.G. et al., "A Provably Secure Operating System: the System, its Applications, and Proofs," Final Report, SRI International, Menlo Park, Calif. (February 1977).

OWIC79 OWICKI, S.S., "Specifications and Proofs for Abstract Data Types in Concurrent Programs," pp. 174-197 in Program Construction, ed. F.L. Bauer and M. Broy, Springer Verlag Lecture Notes in Computer Science, Vol. 69 (1979).

PADL79 PADLIPSKY, M.A., BIBA, K.J., and NEELY, R.B., "KSOS - Computer Network Applications," AFIPS Conference Proceedings Vol. **48**, pp.373-381 (1979). National Computer Conference.

PARK76 PARKER, D.B., "Computer Abuse Perpetrators and Vulnerabilities of Computer Systems," AFIPS Conference Proceedings Vol. **45**, pp.65-73 (1976). National Computer Conference.

PARN72 PARNAS, D.L., "A Technique for the Specification of Software Modules with Examples," CACM Vol. **15**(5), pp.330-336 (May 1972).

PETE81 PETERSON, G.E. and STICKEL, M.E., "Complete Sets of Reductions for some Equational Theories," JACM Vol. **28**(2), pp.233-264 (April 1981).

POPE74a POPEK, G.J., "A Principle of Kernel Design" in "A Panel Session - Security Kernels" (Chaired by S.B. Lipner), AFIPS Conference Proceedings Vol. **43**, pp.973-980 (1974). National Computer Conference.

POPE74b POPEK, G.J. and KLINE, C.S., "Verifiable Secure Operating System Software," AFIPS Conference Proceedings Vol. **43**, pp.145-151 (1974). National Computer Conference.

POPE78a POPEK, G.J. and KLINE, C.S., "Issues in Kernel Design," AFIPS Conference Proceedings Vol. **47**, pp.1079-1086 (1978). National Computer Conference.

POPE78b POPEK, G.J. and FARBER, D.A., "A Model for Verification of Data Security in Operating Systems," CACM Vol. **21**(9), pp.737-749 (September 1978).

POPE79 POPEK, G.J. et al., "UCLA Secure UNIX," AFIPS Conference Proceedings Vol. **48**, pp.355-364 (1979). National Computer Conference.

ROBI77 ROBINSON, L. and LEVITT, K.N., "Proof Techniques for Hierachically Structured Programs," CACM Vol. **20**(4), pp.271-283 (April 1977).

ROBI79 ROBINSON, L., Quoted by P. Zave in "Report of a Panel Session from Specifications of Reliable Software Conference", ACM Software Engineering Notes Vol. **4**(3), pp.17-18 (July 1979).

ROSE73 ROSEN, B.K., "Tree-Manipulating Systems and Church-Rosser Theorems," JACM Vol. **20**(1), pp.160-187 (January 1973).

ROUB77 ROUBINE, O. and ROBINSON, L., "SPECIAL Reference Manual - 3rd Edition," CSG-45, SRI International, Menlo Park, Calif. (1977).

RUSH81a RUSHBY, J.M., "The Design and Verification of Secure Systems," Proceedings 8th ACM Symposium on Operating System Principles (To appear December 1981). (Also Internal Report SSM/7, Computing Laboratory, University of Newcastle upon Tyne, England, April 1981).

RUSH81b RUSHBY, J.M., "Proof of Separability - a Verification Technique for a Class of Security Kernels," Internal Report SSM/8, Computing Laboratory, University of Newcastle upon Tyne (April 1981).

RUSH81c RUSHBY, J.M., "Specification and Design of Secure Systems," in State of the Art Report: Systems Design, ed. P. Henderson, Pergamon Infotech (to appear 1981). (Also Internal Report SSM/6, Computing Laboratory, University of Newcastle upon Tyne, England, April 1981).

SCHA77 SCHAEFER, M. et al., "Program Confinement in KVM/370," Proceedings ACM '77, pp.404-410 (1977).

SCHI75 SCHILLER, W.L., "The Design and Specification of a Security Kernel for a PDP-11/45," ESD-TR-75-69, Mitre Corporation, Bedford, Mass. (March 1975).

SHAN77 SHANKAR, K.S., "The Total Computer Security Problem: an Overview," IEEE Computer Vol. **10**(6), pp.50-73 (June 1977).

SHOE69 SHOENFIELD, J., Mathematical Logic, Addison Wesley, New York (1969).

WALK80 WALKER, B.J., KEMMERER, R.A., and POPEK, G.J., "Specification and Verification of the UCLA Unix Security Kernel," CACM Vol. **23**(2), pp.118-131 (February 1980).

WILK81 WILKINSON, A.L. et al., "A Penetration Study of a Burroughs Large System," ACM Operating Systems Review Vol. **15**(1), pp.14-25 (January 1981).

WOOD79 WOODWARD, J.P.L., "Applications for Multilevel Secure Operating Systems," AFIPS Conference Proceedings Vol. **48**, pp.319-328 (1979). National Computer Conference.

WULF76 WULF, W.A., LONDON, R.L., and SHAW, M., "An Introduction to the Construction and Verification of Alphard Programs," IEEE Trans. on Software Engineering Vol. **SE-2**(4), pp.253-264 (December 1976).

ANON78a ANON., "KSOS Verification Plan," WDL-TR7809, Ford Aerospace and Communications Corp., Palo Alto, Calif. (March 1978).

ANON78b ANON., "Secure Minicomputer Operating System (KSOS) Non-Kernel Security Related Software Computer Program Development Specifications (Type B5)," WDL-TR7934, Ford Aerospace and Communications Corp., Palo Alto, Calif. (September 1978).