Project MAC September 5, 1975

Computer Systems Research Division Request for Comments No. 87

Ph.D. Thesis Proposal: A Methodology for Designing Certifiably Secure Computer Systems

by Richard J. Feiertag

Abstract:

The ability to certify that a computer system operates in a secure manner is highly desirable. A method of designing complex computer systems that will make them easier to certify as secure is proposed. This method involves isolating those parts of the system having to do with security into a "kernel" subsystem and organizing this kernel into functionally oriented program "regions" whose relationships to one another are highly structured. The careful structuring of the kernel allows it to be easily certified. In order to demonstrate the usefulness of this method it will be used to design a computer system that is secure. Problems arising in applying the method to real systems will be explored. The technique for certifying the security of the system will be demonstrated using the example system.

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

Overview

The main purpose of the thesis is to describe and demonstrate a particular method of designing a certifiably secure computer system. One essential part of the method is the use of a particular structure for designing a system. The main feature of the structure is the "kernel" that includes the part of the system that must be correct in order to assure the security of the system. The kernel itself is divided into three hierarchical levels, each level building on lower levels to provide a more sophisticated type of security. The choice of three levels is somewhat arbitrary, however the hierarchical nature of the levels is an intrinsic property of security.

The thesis will include, as a case study, the design of a sample system for which the correctness of the lowest level of the kernel can be certified. The thesis will show that the lowest level is the only level that can reasonably be certified by existing techniques. The lowest level will be divided into small, functionally oriented modules. The use of functional orientation for separating modules is one means of minimizing the amount of interaction between modules and, therefore, eases the certification process. Although most of the above ideas are not new, this is the first time they have been applied together to a design and certification of a practical operating system.

The overall design of the modules is made without regard to the location of the hardware, firmware, and software boundaries of the system. The determination of the hardware, firmware, and

software boundaries is largely an efficiency and economic decision and should be delayed until later in the design process. These boundaries can, therefore, occur in the middle of modules and methods of certifying modules that are partly hardware, firmware, and software will be described.

Some difficult problems arise when using a high level language for programming the kernel. Some programs will have to run in a very primitive environment and others will run in heavily embellished environments. It is important that the certifier, programmer, and compiler know the precise nature of the environment of the program he, she, or it is working on. High level languages tend to isolate the programmer from knowledge of the precise nature of the environment in which the programs operate. This is especially troublesome during system initialization when environments may be only partly operational. The thesis will describe a scheme by which a program's environment will be obvious to all concerned. The scheme will force the programmer to include in the programs he writes a description of the environment he expects the program to run in. The compiler will be able to check at compile time that the environment specified for the program it is compiling will be available to the program when it runs.

In a general purpose operating system, there is no guarantee that a user written program running outside the kernel will adhere to the conventions of the language used to implement the kernel. This presents a problem at the kernel interface. A

scheme will be described that guarantees that the kernel cannot be led astray by improper calls. In this scheme the compiler and the call mechanism will together assure that each call into the kernel from outside the kernel will be made only to defined kernel entries and that the arguments passed to the kernel are those which the kernel expects (i.e. the arguments it is certified to handle correctly). The methodology will also include a means of protecting the kernel programs and data from modification by programs outside the kernel. It will be demonstrated that this protection scheme is at least as desirable for protection of the kernel as others in use in other operating systems.

One of the objectives of the methodology is to place much of the burden of certification on the methodology itself rather than on the system designer. The methodology need only be certified once and then can be used in the construction of many different systems.

The overall goal of the thesis is to demonstrate that the method described can be used to design a secure operating system that is highly readable, understandable, and easily modifiable and yet is useful and efficient. To accomplish this the thesis will include a complete design for the lowest level of the kernel of a system that implements a useful subset of the features of the Multics operating system.

Introduction

An extremely desirable property of a computer system is that

it operate properly. This vague statement can be interpreted to mean that the computer system operates in a manner consistent with the intent of its designers and users. Much effort is currently being directed towards finding ways of assuring that a computer system behaves as intended. I will use the word *certification* to express the general idea of assuring that a computer system does what was intended.

Another desirable property of a computer system is that it be secure. A secure system is one that allows a user to maintain information on the system in privacy, i.e. other users cannot acquire or modify the information unless authorized to do so. A secure system is also one whose integrity is assured, i.e. the system cannot be tampered with improperly. A more precise definition of security will be given later.

Both proper and secure operation are necessary if a computer system is to function as a utility serving many different users of diverse needs on a regular basis. Being able to build a computer system that can be demonstrated to be secure, i.e. is certifiably secure, is a necessary step to providing a true computer utility. The problems of certifying a system and the problems of system security will be carefully examined and a method for building a certifiably secure system will be proposed.

The first part of this proposal briefly outlines current certification techniques and describes some methods for designing more easily certifiable systems. The advantages and disadvantages of these methods are discussed. A new method is

proposed and justified. The second part of this proposal discusses security and the difficulties of building a truly secure system. A definition of security is outlined as a goal for certification.

To demonstrate the feasibility of this new method of designing a certifiable system, a case study is proposed. The case study involves the design, using the proposed method, of a certifiably secure system. The overall plan for the design is outlined. The certifiability of the system resulting from the design will be demonstrated in the thesis.

System Certification

Certification of a system can be divided into two parts. First, the intent of the designers is expressed in a formal language and the result is called the specification of the system. Second, one must "prove" that the programs and hardware operate in a manner consistent with the specification. In reality, one would expect the two parts of certification to be an integrated iterative process. The work to be described here deals mainly with the second part.

Proving that the software meets the specification is called program verification. In program verification a formal mathematical description of a programming language is derived. A specification, in terms of mathematical assertions about the relationships between input and output variables of the desired program, is defined. Then the program in question, written in the formally described language, is shown to logically imply the

assertions of the specification. These techniques are described and demonstrated by Elspas et al. [El72] and surveys of the area are given by Elspas et al. [El72] and London [Lo72]. Unfortunately, success in this area is limited to very simple programs (e.g. sorting, mutual exclusion) with very simple specifications. Even in these cases the proofs are nontrivial. Some larger programs have been verified, but these proofs require tremendous amounts of effort. If one were to scale these efforts up to the size and complexity of a sophisticated operating system, the effort would be staggering, if not impossible. The main hope in this area lies in automation. Igarashi, London, and Luckham [Ig73] have developed some semi-automated techniques for program verification by computer. At this point these techniques can be applied only to fairly simple programs in a restricted language and are not always successful. Certification of entire systems by these techniques is still a long way off. If there is to be any means of certifying systems in the near future it will be necessary to somehow divide the system into much smaller and less complex pieces that can be verified separately by contemporary techniques.

Once one has divided a system into smaller pieces so that each piece may be more easily verified, one introduces the new problem of demonstrating that the individually verified pieces together form the desired whole. One would expect that this new problem is easier to solve than the original problem of certifying the entire system. However, for an arbitrary division

into pieces, there is no guarantee that the new problem is easier. The designer must carefully divide the system in such a way as to simplify the task of demonstrating that it forms the desired whole. For ease of certification, one would like to have these pieces or modules interact with each other as little as possible to limit complexity and to make the description of each module as simple as possible. In Parnas' [Pa72M, Pa72D] "information hiding" approach, the description of each module states only what is necessary to provide the stated service of that module. Any other features that may exist due to the implementation of the module are not described. The Parnas technique encourages the simplest possible description of each module and also allows the greatest flexibility in the implementation of the module. The thesis will show that these desirable properties (simple description and flexibility in implementation) of the "information hiding" approach are also present in the design presented in the thesis.

Clearly, the choice of modularization is critical to the success of a certification attempt. The main goal of the thesis will be to describe, justify, and illustrate one particular methodology for modularizing a computer system so that it may be easily certified.

Structural Approaches Used by Others

Several methodologies for modularization appear in the literature. One such approach, called "levels of abstraction", was originally described by Dijkstra [Di68] and more recently

used by Neumann et al. [Ne74]. In this approach, the programs of the system are divided into a set of disjoint levels. The bottom level implements the most primitive functions and uses only the machine hardware. Higher levels implement more complex functions and may utilize the functions of lower levels as well as the hardware. The uppermost level implements the functions of the entire system by utilizing all the functions implemented in the other levels. In this technique the levels are totally ordered, i.e. each level utilizes, and therefore is dependent upon, only functions implemented in lower levels. This total ordering provides an obvious proof technique. The designers must write a specification for each level. One verifies each level by verifying the correctness of programs in that level using the specifications of the lower levels. The lowest level can be directly verified since it is not dependent upon any other levels. If the levels are properly designed, the verification of each level should be relatively straightforward. The major drawback of this technique is the difficulty of modularizing a system into a totally ordered set of levels. In existing complex systems many system functions are mutually dependent and it is not clear that they can be totally ordered. In these systems a total ordering apparently requires a modularization along artificial lines and this can require splitting functions among several levels and duplication of mechanisms in several levels. In the Dijkstra "THE" system, the system is very simple and the total ordering is easily achieved. However, it is not clear that

a straightforward total ordering exists for larger systems. The large system described by Neumann demonstrates the problem. It necessitates splitting the same function among several levels. His virtual process manager and virtual storage manager are so divided. This artificial splitting is undesirable because the recurrence of the same conceptual function in different levels makes the system harder to design, harder to understand, and harder to modify. It is more desirable to have each function implemented at only one level. Neumann has, therefore, not yet demonstrated that a total ordering is a optimal approach.

Another approach is described by Wulf et al. [Wu74]. In this case, a small group of functions fundamental to the operation of a computer system are isolated. These functions are collectively called the "kernel" and form a small central core of the computer system. Wulf assumes that since the kernel is small it can be verified by present techniques of program verification. This approach is quite useful as far as it goes. The Wulf kernel is designed to be a central core around which many operating systems can be built and run simultaneously on the same machine. For this purpose it may serve well. However, this kernel is not an operating system itself and cannot be used as such. The systems built around the kernel must still be certified and the same problems of size and complexity still remain.

The approaches to modularization of Neumann and Wulf have been discussed in the context of software certification. Such modularization techniques can work equally well with hardware

certification. The general view of the author is that these techniques apply to systems as a whole without distinquishing between hardware and software. Whether a particular algorithm is implemented in hardware or software is not of concern in these modularization techniques. Of course, the verification techniques are different for hardware. Instead of programs, the correctness of wiring diagrams must be verified.

This thesis will not attempt to advance the state of the art of verification of hardware or software. It will be assumed that such verification techniques do exist and can be applied to programs and circuits of moderate size. The thesis, instead, will concern itself with the problem of using a good modularization technique to divide large complex systems into programs and circuits that can be certified by such verification techniques, and to demonstrate that the modules so derived, when collected together, can be shown to form the desired whole. Even if it should be the case that verification techniques are not sufficiently advanced to be applicable to the programs resulting from the modularization techniques to be proposed, the modularization techniques will still be of benefit in allowing readers of the system to better understand the system and have increased confidence in its correct behavior. If mathematical proof techniques cannot be used, then simple review of programs by impartial observers might be a viable alternative in many situations. Although not nearly as reliable as mathematical proof, the manual review of code by observers can uncover many

errors and provide a degree of confidence in the correctness of the code sufficient to satisfy at least some customers. Proper modularization will make the understanding of code much easier and give an observer the ability to say that he or she understands the system, a feat not possible in the large systems of today. The important point is that proper modularization is important no matter which method (mathematical proof or manual reading) is used to verify programs. Either program verification method is made easier by proper modularization. The choice of program verification method can be made largely independently of the choice of modularization technique.

The Kernel Approach

The modularization technique to be proposed here also involves designing a kernel. However, the criterion used for deciding what should be in the kernel is different from that of Wulf. Before stating this criterion I must observe that in a computer system that is to be certified, it is probably true that not every program or line of code is critical for meeting the specification. No matter how these noncritical portions are modified, although the operation of the system may change in some way, the system will continue to meet the specification. These noncritical portions of code arise because a specification does not necessarily cover all aspects of the operation of the system. (1) For example, a system may be specified to contain virtual

---

(1) For the purpose of this paper I use the word specification to include the definition of only those properties of the system which the designer wishes to assure the presence of by

processes and that each process be guaranteed to run for a total of at least one minute in each hour. Such a specification leaves unstated most of the scheduling algorithm, e.g. the order in which the processes will run, whether each process will run for one continuous minute or shall be run several times within the hour for shorter periods, etc. The scheduling algorithm of the system must be known if the system is to be built, but all of it need not be verified. The scheduling algorithm can be constructed so that one can verify that each process gets its minute under all possible circumstances, without having to verify other properties of the scheduling algorithm. One such scheme might have a special program that, at the time of each scheduling, checks to see if at the time of the next scheduling there will be enough time left in the current hour to meet the specification for all processes. If such is not the case this special program preempts the regular scheduling algorithm and schedules the deprived processes. In this case only the special program need be verified.

In any system, for any given specification, it is likely that much, if not most, of the code is noncritical. Once the critical code is isolated, only this code need be verified. A <u>kernel</u> is the critical code of a system, i.e. that part of the system that must be verified in order to certify that the system meets a specification. This kernel is not the same as Wulf's and

---

certification. Such a specification might not include more general information that must be known in order to use the system, such as calling sequences for library trigonometric functions.

all future uses of the term will refer to this new definition. Clearly, one can write a specification for a system in which all code is critical. In such a system, defining a kernel is not useful because the kernel will include the entire system. However, in many applications it is desired that only certain aspects of system behavior be verified. Security is an example of such an aspect of system behavior. To many organizations, the privacy of information is more important than accessibility to or correctness of the information. In a multiplexed computer system, a user can usually provide his own means of assuring the correctness of information, but the security of the information can be assured only by the system. Therefore, it might be considered more important to certify the security of the system than to certify that the system processes user information correctly. One of the main objectives of the thesis will be to show that code critical to security can be isolated and that it is a small subset of the code in a typical operating system. The thesis will also discuss other situations in which a kernel cannot be isolated. This will help identify those cases in which defining a kernel is a useful approach to certification. Although defining a kernel is not a completely general approach, it is applicable to some important cases.

I have, above, talked about critical _lines_ of code. Program verification techniques are easily applicable only to entire programs, not to individual lines. Unfortunately many programs in a system are likely to have only a few critical lines. To

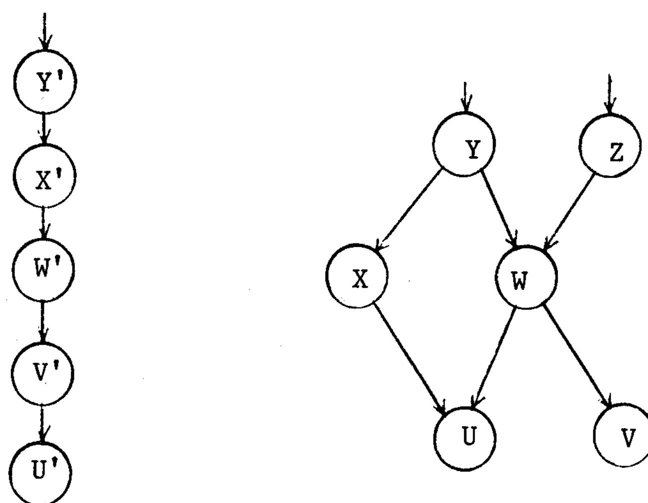
verify these few lines it would be necessary to include the entire program into the kernel, thereby making the kernel much larger than it has to be. Unnecessary increase in size makes verification more difficult. For this reason, it is imperative that the system be designed to isolate the critical code into separate programs. In fact, it is to the designer's advantage to first write the kernel specification. The designer can then demonstrate the correctness of the kernel specification by proving that it implies the system specification and then write and verify the kernel programs. This is the approach taken by Bell and LaPadula [Be73,LP73] and Walter et al. [Wa74].

The Kernel Infrastructure

The above approach is useful in the case where the kernel is sufficiently smaller than the entire system to justify the additional burden of defining and proving the correctness of a kernel specification. However, even if such is the case the kernel may still be too large to verify by present techniques. A first thought might be to find a subkernel for the kernel, however, if one could find such a subkernel smaller than the kernel it would mean the kernel was not defined properly in the first place. Another possible method is to use the "levels of abstraction" approach mentioned earlier. This approach applies just as well to a kernel as it does to a whole system. Unfortunately the disadvantages still remain. To alleviate the disadvantages, I propose a two stage generalization to the "levels of abstraction" approach. These generalizations relax

some of the restrictions on the structure of the system.

The first generalization is to permit the levels to be partially ordered rather than totally ordered. In a partially ordered hierarchy, the word "level" is not appropriate and the word <u>region</u> will be used instead. An example of a schematic representation of a totally ordered and partially ordered system structure is:



Total Ordering Partial Ordering

Fig. 1

Circles represent regions (or levels). The relation (U)→(V) means that region U is dependent upon (higher than) region (level) V, i.e. that U is dependent upon some service provided by V in order to meet its specification. The partial ordering retains all the advantages of the total ordering in that there is still an obvious proof technique. The designer writes a specification for each region and verifies the correctness of the region by verifying the correctness of programs in that region using the specifications of other regions upon which the region in question is dependent. Regions which are not dependent on any

other regions, i.e. regions with no arrows leaving, are verified by proving the correctness of the programs in the region. The verification process is straightforward because, as with total ordering, there are no directed loops. The partial ordering structure is more useful than the total ordering because the former displays more structural information about the kernel. Consider the case:

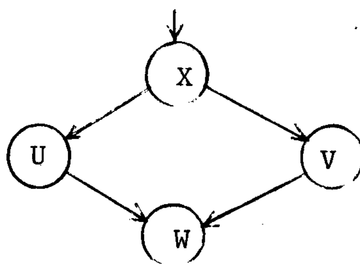


Fig. 2

If this structure were designed using the total ordering technique some likely structures would be:

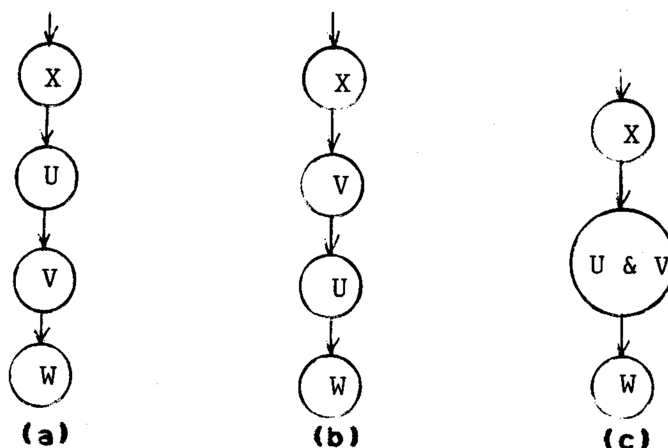


Fig. 3

In the first two total orderings the relationship between U and V is extraneous. U and V are actually unrelated, but the total ordering constraint forces this unnecessary relationship. In case (c), the distirction between regions U and V are lost and

one large level must be verified rather than two small independent regions. The partial ordering clearly shows the true relationship. Such a relationship cannot be expressed in a total ordering. One can argue that this advantage is insignificant in practice, but any representation that clarifies the structure of the system is useful for purposes of certification.

The second generalization is to allow for directed loops in the structural graphs. Directed loops are undesirable because they make verification difficult. If two regions are mutually dependent upon one another then there are no obvious regions upon which to base a verification. Without directed loops the starting points are the nondependent regions. For this reason directed loops are to be avoided. However, directed loops do occur in actual system designs, usually in the form of mutually recursive programs. Rather than try to design away directed loops as does Neumann, the technique proposed here will allow directed loops but will try to limit the amount of added complexity to the verification process. This will be done by designing the system so that the number of regions contained in a directed loop is minimal. For the purpose of minimizing the number of regions in a directed loop, the partial ordering technique is superior to the total ordering technique, because, as shown earlier, the partial ordering technique more clearly shows the true dependency relations between regions without adding extraneous dependencies. For example, consider Figures 2 and 3a again with a directed loop between U and W:

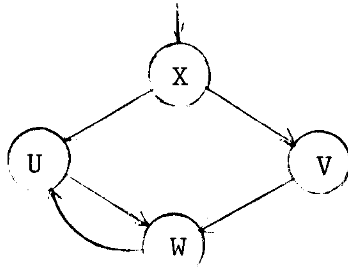


Fig. 4

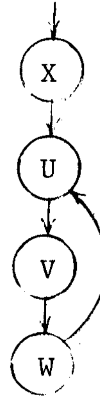


Fig. 5

In Figure 4 it is clear that the directed loop involves only regions U and W, whereas, in Figure 5, V is apparently part of the loop even though in reality it is not involved.

This technique does not make the verification of systems with mutually recursive programs easier to verify, but it does provide a means of indicating the relative difficulty of verifying a particular directed loop. In those cases where a directed loop involves a large number of regions, it might be advantageous to try to eliminate or reduce the size of the loop by some redesign. However, in some cases the cost of a redesign may be greater than the cost of verifying the correctness of the directed loop. By allowing for directed loops, the technique proposed here recognizes that there is an engineering tradeoff and provides some means for evaluating that tradeoff.

The concepts of kernel and region suggest the methodology to be explored in the thesis. Once a system specification has been determined, the algorithms and data essential for the fulfillment of the specification are identified and a kernel specification is written. The algorithms and data of the kernel are then subdivided into regions and specifications are written for these

regions. The designing of the kernel and the region specifications is, undoubtedly, not a strictly sequential process, but is more likely to be an iterative process with successive refinements being made to all specifications as problems become better understood. The concept of a kernel serves to minimize the amount of code that must be dealt with. The regions serve both as a structuring technique to keep the system organized and to further reduce the size of the units which must be verified. With both the concept of the kernel and the concept of regions serving to limit the size of unit of verification, i.e. the region, the resulting regions should be small enough to be verified by either mathematical proof or manual review techniques.

In addition to understanding the inherent advantages and disadvantages of the structures presented above, it is necessary to investigate the ease with which these structures can be applied to large computer systems. If a system designed using the structure described above is to be comprehensible and certifiable, the regions and dependencies of the design should easily identifiable as modules and interactions between modules in the system. In software, regions can be identified as programs or collections of programs. However, in most conventional programming languages, the interactions between programs is not always easily identifiable and even when identified, the interactions can be difficult to describe. For example, several programs may interact by sharing a variable

common to all. In order to describe this interaction, one may have to know how each program can modify the value of the variable and when each program will reference the variable relative to the other programs. Since interactions such as these usually constitute dependencies between regions, the interactions must be fully specified if the overall structure of the kernel is to be understood.

In order to facilitate the identification and description of interactions between regions, it is desirable to use a programming language that permits only a few well defined means of interaction between programs. Clearly, the means of interaction that are permitted by the language must be properly chosen so as not to make the necessary interactions clumsy or inefficient. One such language is CLU [LI74]. CLU defines a program unit called a *cluster*. Clusters may interact only by means of procedure calls. Implementing regions as clusters makes identification of interactions between regions easy, i.e. all interactions between regions are simply procedure calls from one cluster to another. For this reason, the thesis will use the CLU language for programming examples of regions. The thesis will also discuss other language features and system features in general that will make regions and dependencies between regions easier to identify and describe.

System Security

At least some minimum security is essential in a certifiable system. To certify a system one must not only prove that the

programs operate correctly, one must also demonstrate that the programs cannot then be improperly modified. Such protection is a form of security. Also security is necessary to insure privacy of information and reliable operation of the system. For these reasons, among others, security must be considered fundamental to the operation of any large certified computer system. Assuring system security is of great general interest among system designers and users because of the desire to insure the privacy and longevity of information.

One common definition of a secure system is one in which there can be:

1. no unauthorized release of information,
2. no unauthorized modification of information,
3. and no unauthorized denial of service.

There are some key words in this definition of security. *Information* refers to data stored in files by users of the system, data stored by the system itself, and data about the operation of the system. Examples of information are: a user stored list of names and addresses in a file in the system to be used as a mailing list, a system list of users who are currently permitted to use the system, and whether or not the system is currently running. *Service* is any operation which the system provides. The ability to log into the system, to use I/O devices, and to communicate with other users are examples of services provided by many systems. Clearly, what constitutes a system service depends upon the system. *Authorization* is a set

of rules or algorithms which determine which requests for operations to be performed by the system are to be honored. Again, what constitutes authorization is dependent upon the particular system. Some common types of authorization mechanisms are capabilities, access lists, and security levels and categories. The choice of a form of authorization is crucial to determining how usefully secure a system will be. The choice that all requests for operations be honored will hardly lead to a usefully secure system. The authorization policy must, therefore, be consistent and produce some minimum level of protection, i.e. enough protection to insure the integrity of the system programs themselves. The issue of what constitutes a meaningful authorization policy will be discussed in the thesis.

The above definition of security, being very general, appears to include all the security which a designer would wish to include in a computer system. However, it is the author's belief that this definition is too general and too strong and that it is not achievable in any practical operating system. This belief is due to what Lampson has described as the confinement problem [La73]. Lampson describes a solution to the confinement problem. This solution requires enumeration of all information paths in the system and the suitable restriction of information flow along these paths. The thesis will attempt to demonstrate, by discussing the difficulties, that the enumeration of all information paths in any nontrivial computer system is impractical, if not impossible. Lampson's solution is not

workable. Therefore, there is no known way of achieving security, as defined above, in operating systems which attempt to simultaneously support more than one user with information he or she wishes to keep private. I will define a workable definition of security less strong than the one given above. Using this new definition of security, the security kernel and the regions within that kernel will be specified for a practical operating system.

The choice of security as the object of certification is advantageous because a system must have secure programs to be certifiable and security is an important property of operating systems. There is currently much interest in building secure systems. Specifying a security kernel also has one major problem. Security is a pervasive property of a system. Every resource or object provided by the system for use by other programs will, in general, need some kind of security. This will mean that the security kernel may not be so small. However, it does provide a natural means for further modularizing the security kernel into regions, i.e. each module being responsible for the security of one type of object or resource, and this modularization will be employed in the thesis.

The Case Study

The actual specification of a security kernel and its regions is important. The actual specification of a kernel and its infrastructure will demonstrate the practicality of this approach to certification. That is, it will demonstrate whether

or not a small kernel and its constituent regions can be easily identified and implemented and, if so, whether or not the resulting regions are small enough and simple enough to be verified by mathematical or manual review techniques.

The thesis will, therefore, include a case study. A design for a security kernel for a practical operating system will be performed. The operating system for which this security kernel is to be designed is a mildly abstracted form of Multics [Or72]. The choice of an actual operating system is necessary to avoid designing a toy example in which many real problems have been abstracted away. The use of a mild abstraction, however, is necessary to avoid having the thesis mired down in discussions of details and idiosyncrasies of the actual Multics implementation. The kernel specification produced will, however, be directly applicable to Multics.

There are several reasons for making Multics the system of choice:

1. It is believed that the Multics architecture is basically secure and that there are no fundamental flaws in its security design. Many other large operating systems would require very fundamental changes to make them secure.
2. Multics is a commercially viable system, having actually been marketed.
3. Multics is a large and complex system. It is a general purpose system with wide applicability and a full range of functions.

4. Multics is well documented and the documentation is generally available.

5. The author is intimately familiar with Multics.

6. It is possible that some of this work may be incorporated into Multics.

7. The facilities of Multics are sufficiently general that parts of the specification of a security kernel for Multics will be directly applicable to other systems.

The security kernel specification will concentrate on the objects: files, processes, and I/O devices, since these objects are common to most operating systems in some form. It is these objects that must be made secure in order to have a secure operating system. As stated above these protected objects provide a natural means for modularizing the kernel, that is, for each type of object, there will be a region in the kernel whose purpose it is to provide the security for that object. A specification for each region will be described. For example, one of the most fundamental objects in Multics is the page of data. The thesis will show that the following six functions (2) implemented in the kernel will assure the security of the data in a page:

<u>read</u> (PAGE , offset)
<u>write</u> (PAGE , offset , value)
<u>make copy</u> (PAGE.COPY(n) , FRAME)
<u>delete copy</u> (PAGE.COPY(n))
<u>set current</u> (PAGE.COPY(n))
<u>remove current</u> (PAGE.COPY(n))

---

(2) A detailed description of the six functions will be given in the thesis. They are included here only to give the reader an idea of the size and complexity of a region.

The programs necessary to implement these functions constitute a region of the kernel. The definition of the six functions are the specification of the region.

Dependencies between regions develop when some objects are constructed using more primitive objects. In Multics, segments are collections of pages, so the region that assures the security of segments will be dependent upon the region responsible for pages. Specifications, in terms of definitions of the functions provided by the regions, will be derived for regions concerned with the security of files and I/O devices in Multics. The specification for the page region, the calls of which are given above, is fairly simple and the design will attempt to maintain this level of simplicity for the specification of all regions. Programs for the regions will also be written using the CLU language.

Conclusion

The main goal of this thesis is to demonstrate a new methodology for designing certifiably secure operating systems. A new technique has been proposed for structuring a system design so that it may be more easily certified. This proposal has argued that this new technique has advantages over other techniques that have been proposed. The thesis will include a case study which applies this technique to a practical operating system. This case study will demonstrate the applicability of the technique.

To limit the scope of the thesis to a reasonable level, I

have chosen to concentrate on certification of one specific property of a system, namely security. This particular property was selected because of its importance to users of the system and because some security is necessary in a system if it is to be certified. A definition of security that is both useful and workable will be described more fully and formalized in the thesis.

The case study will derive a design, using the new technique proposed above, for a secure system similar to Multics. The design will describe the kernel for a Multics like system, giving specifications for the regions which comprise the kernel. Programs that implement these regions should be small enough so that contemporary verification techniques can be used to demonstrate their correctness. The thesis will also show how the specifications for the regions can be used to prove the security of the kernel and the system.

Security is one property of a system that appears well suited to certification using the design methodology proposed. It is not clear that this methodology is equally well suited to certifying other properties of systems. The limitations of the methodology will be explored; examples will be discussed and an attempt will be made to define the bounds of its practical application.

The major results of the thesis will be a general method for designing certifiably secure systems and a design for a specific certifiably secure system. The general method will also be

applicable to designing systems about which properties other than security can be certified.

BIBLIOGRAPHY

[Be73] Bell, D.E. and LaPadula, L.J., *Secure Computer Systems: Mathematical Foundations*, MTR - 2547, Vol. I, MITRE Corp. March 1973

[Di68] Dijkstra, E.W., "The Structure of the "THE' - Multiprogramming System", *Comm. ACM*, Vol. 11, No. 5, May 1968, pp. 341-346

[El72] Elspas, B., Levitt, K.N., Waldinger, R.J., and Waksman, A. "An Assessment of Techniques for Proving Program Correctness", *Computing Surveys*, Vol. 4, No. 2, June 1972, ACM, pp. 97-147

[Ig73] Igarashi, S., London, R.L., and Luckham, D.C., *Automatic Program Verification I: A Logical Basis and Its Implementation*, Stanford Artificial Intelligence Laboratory, Memo AIM-200, May 1973

[La73] Lampson, B.W., "A Note on the Confinement Problem", *Comm. ACM*, Vol. 16, No. 10, Oct. 1973, pp. 613-615

[Li74] Liskov, B., "A Note on CLU", Computation Structures Group Memo 112, Project MAC, M.I.T., November 1974

[LP73] LaPadula, L.J. and Bell, D.E., *Secure Computer Systems: A Mathematical Model*, MTR-2547, Vol II, The MITRE Corp., May 1973

[Lo72] London, R.L., "The Current State of Proving Programs Correct", *Proc. of ACM Annual Conf.*, ACM, 1972, pp. 39-46

[Ne74] Neumann, P.G., Fabry, R.S., Levitt, K.N., et al., "On the Design of a Provably Secure Operating System", Stanford Research Institute Computer Sciences Group, Menlo Park, California

[Or72] Organick, E.I., *The Multics System: An Examination of its Structure*, M.I.T. Press, Cambridge, Mass., 1972

[Pa72M] Parnas, D.L., "A Technique for Software Module Specification with Examples", *Comm. ACM*, Vol. 15, No. 5, May 1972, pp. 330-336

[Pa72D] Parnas, D.L., "On Criteria to be Used in Decomposing Systems into Modules", *Comm. ACM*, Vol. 15, No. 12, Dec. 1972, pp. 1053-1058

[Wa74] Walter, K.G., Ogden, W.F., Rounds, W.C., et al., *Primitive Models for Computer Security*, Department of Computer and Information Sciences, Case Western Reserve University, Jan. 1974

[Wu74] Wulf, W., et al., "HYDRA: The Kernel of a Multiprocessor Operating System", Comm. ACM, Vol. 17, No. 6, June 1974, pp. 337-345