PARTITIONS AND PRINCIPLES FOR SECURE OPERATING SYSTEMS

Gregory R. Andrews

TR 75-228

February 1975

Department of Computer Science
Cornell University
Ithaca, NY 14853

PARTITIONS AND PRINCIPLES FOR SECURE OPERATING SYSTEMS

Gregory R. Andrews

ABSTRACT

As part of the general goal of providing secure computer systems, the design of verifiably secure operating systems is one of the most important tasks. This paper addresses the problem by defining security in terms of a model and proposing a set of principles which we feel should be satisfied in a secure operating system. Informally, an operating system is secure if its users completely control the use of all information which they introduce. Four key partitions are identified: user interface functions, user invoked services, background services, and the security kernel. Principles are then defined to insure that interface functions provide a safe initial environment for executing user programs, user called services are confined, background services have no access to user information, and the security kernel adequately protects information storage.

PARTITIONS AND PRINCIPLES FOR SECURE OPERATING SYSTEMS

Gregory R. Andrews

INTRODUCTION

As more information has been placed in computer systems, society's concern for controlling its use has greatly increased. In response to this concern, numerous groups are actively engaged in various aspects of identifying security goals and designing secure computer systems [4,10,13,14,15,16,18,19]. Providing security requires both that system software carries out the intent of users as expressed in programming languages and that the hardware and physical environment do not introduce errors or provide paths for information to leak (e.g. via wiretapping). This paper will focus on the first problem by describing an approach to the design of secure operating systems.

We view an operating system as having four distinct components with respect to security:

(1) interface functions, such as job management or user identification, which communicate directly with the external user;

(2) user invoked services, such as the file system;

(3) background services, such as storage managers or checkpoint generators; and

(4) the protection system (security kernel) which provides mechanisms for controlling access to stored information.

Informally, we consider such a system to be secure if its users completely control the use of all information which they introduce into the system. Our specific aim in this paper is to present a model and definition of operating system security and then to propose a set of principles for the above partitions which might lead to a provably secure system.[1] In particular, we are concerned with safely interfacing users to the system, confining services they invoke, isolating other services from their information and insuring the adequacy of the system's protection mechanisms. The ideas presented are a synthesis and extension of previous work and are intended to illustrate our approach; preliminary results indicate that this approach looks promising.

A MODEL OF SOFTWARE SECURITY

The role of any software system is to interpret and carry out the commands of its users. For this purpose an operating system (OS) provides a job control (command) language and a variety of programming languages. In these languages, users describe the actions they wish to perform and also describe actions which they will allow other users to take on their programs or data. The first component of our security model is a characterization of the control a user might express in a language; we call this our external model. The role of the OS is to carry out the user's actions by mapping his information into physical resources and providing services for him to use. In order to control access to stored information, an OS employs a variety of protection mechanisms. The second component of our model is therefore a security kernel consisting of data structures and primitive operations used for protection. The role of the operating system is to accurately translate and obey the access restrictions expressed in a language and implemented by the security kernel.

External Model

Three types of external objects are of interest with respect to security: users, user information, and job control statements. Users are individuals having a unique name or other means of identification. User information consists of the programs and data which a user wants processed; part of the information he submits is a set of rights specifying the actions which other users can take on his information. A right in our model is a triple, (user name, information specification, action); it states that the named user can take the action (e.g. execute an operation) on the specified information. For example, a user might say that a subroutine is to be made accessible to all who want to use it or that a collection of research data is to be readable by his colleague named SMITH. The final component of the external model is the set of job control statements by which a user specifies the tasks (job-steps) which the OS is to initiate on his behalf.

---

[1] The presentation is semi-formal, omitting much of the detail needed for actually proving security; when appropriate, relevant formalisms are referenced.

Security Kernel

Externally specified user rights are implemented by the mechanisms of the security kernel. Its role is to control the actions of computations by limiting their access to stored information. Any protection kernel [2,10,20,22,23] can be abstractly characterized by four components:[2]

(1) The set of objects to be protected.

(2) The protection state containing object descriptors and capabilities which define authorized actions.

(3) A set of monitors (enforcers) which determine the legality of every action by consulting the protection state.

(4) A set of protection primitives used to change the protection state.

With respect to protection, three classes of objects exist: actors, information structures, and physical resources. Actors as the name implies are the objects who take actions; examples are processes and procedures. Information structures are logical collections of information, analagous to segments; examples are data segments, code segments, process descriptors, argument lists, messages, files, and file directories. Finally, the resources are the hardware addressable objects such as memory blocks, pages, disk sectors, and devices.

The protection state at any point in time contains descriptors describing each object and capabilities which are associated with actors and define the authorized actions. A capability has the general form: (object name, access attributes, control attributes); it authorizes a set of access actions on the named object and a set of control actions (protection primitives) on capabilities for the object. The access actions are dependent on the type of the object; examples are Call a procedure, Read a segment, or Input from a device. The control actions are type independent and consist of some subset of the protection primitives implemented by the kernel (see below).

The third component of the security kernel, the enforcement mechanism or monitors, is a set of hardware and/or software procedures which validate actions by consulting the capabilities in the protection state. In particular, a monitor consults the domain of the executing actor which contains his local capabilities (ones he always has, such as for local variables in a procedure) and his dynamic capabilities (such as those for objects in an argument list). Examples of monitors are the hardware segment

---

[2]This model is described in detail in [1]; other protection models [6,8,9,11,21] contain similar components.

access validation in segmented machines and the software checks performed when a file is opened.

Because computer systems are dynamic, some means to change the protection state must exist. For this purpose the security kernel contains protection primitives such as Grant a capability, Allocate memory, Create an object or Load a program status word.

OPERATING SYSTEM PARTITIONS

The function of a secure operating system is to correctly map components of the external model into the security kernel. In agreement with others [10,13,14,15,16,22] we feel that it is important that the system's protection mechanisms are collected together in a kernel rather than distributed throughout the OS. In this way, the kernel can be used to provide protection in the rest of the system and the implementation of the kernel itself can hopefully be certified since it is small. Because a typical OS is large and complex, proving it to be correct is beyond our forseeable ability. Therefore it appears necessary to partition the OS, by functions, into smaller components; we can then define the requirements for security and develop provable assertions for each partition (as long as they are disjoint with respect to security). We now expand upon the nature of the three partitions which we feel are appropriate: interface functions, user-invoked services, and background services. Their role in mapping external objects into kernel objects is summarized in Figure 1.

Through job control or command language statements, the external user interacts with four basic interface processes of an OS. The first is the computation which identifies users upon receipt of a job card or log-on statement; it has the crucial role of associating a user with his and only his internally stored capabilities. Second is the job manager or terminal listener which translates command language statements and invokes the requested tasks. The third component, at least in most batch systems, is the spooling subsystem which transfers user information into internal information structures. And the fourth component is the set of language translators which produce executable code from user source code and translate user rights into kernel capabilities.[3] The relevance of these interface processes to security is that each maps some part of the external, user model into internal objects and this mapping must be performed in a way which enables user security to be preserved.

Once a user process (task) has been initiated, it generally invokes a variety of operating system services. These services exist to make it easier for users to compute and to enable them

[3] Language translators are not really part of an operating system but are included here because their translation role is crucial to security; this point will be developed further.

to cooperate; examples are the file system, page managers and inter-process communication primitives. This type of operating system computation is either explicitly called (e.g. file access) or implicitly invoked (e.g. a page fault) by an executing process and is invariably passed some user information. For security, we want to insure that these services do not erroneously or maliciously divulge the information of their caller, except by his explicit request.

The final group of operating system computations we call background services. This class includes system measurement, storage coallescing, and checkpoint generation. These services operate in the background in an attempt to keep the system running reliably and efficiently. They are not directly invoked by users but have an existence of their own. For security, we want to insure that they cannot possibly give information of one user to another.

DEFINITION OF SECURITY

In terms of our model, we can now define our view of security. First, an operating system is secure if each of its users is secure. A user is in turn secure if his information is secure. Let U be the set of users of a system, let each user, U_i, have sets of data, denoted $D_{i,j}$, which he wishes to protect; and, finally, let R be the set of rights which U_i submits to the system.[4] With this notation, information security is defined as follows:

<u>Definition</u>: Information $D_{i,j}$ of U_i is <u>secure</u> if user U_k can take action α on $D_{i,j}$ if and only if $(U_k, D_{i,j}, \alpha) \in R$.

Our intent here is not to present a rigorous definition of security but only to use enough precision to give a clear idea of what we mean; a formal definition is contained in [4] and relevant Air Force work is referenced in [4] and [19]. In words, this definition says that no user, U_k, should be able to operate on the information of another user, U_i, unless U_i explicitly says that U_k has permission. To verify that an operating system is secure, we must prove that no component provides a means by which this definition can be violated.

INTERFACE PRINCIPLES

To insure that the user interface processes are secure, verifying three principles appears sufficient: identification, task initiation, and program translation. It is quite obviously imperative that no user be able to masquerade as another for otherwise the masquerader could exercise rights not given to him. This necessitates our first principle.

---

[4] Recall that each right names a user, some information and an action allowed on the information.

<u>Identification</u>: Each user, U_i, of the OS, is identified uniquely whenever he signs on.

Numerous researchers have examined this problem and advanced possible solutions; for example, see [7,17]. If the code and data of the identification process are isolated from all other computations, then it currently is possible to certify a high degree of effectiveness.

The second crucial operation performed by interface processes is command language translation and the resulting initiation of user requested tasks (job-steps) such as COMPILE or EXECUTE. It is imperative that tasks initiated for one user do not have capabilities for objects of any other user; in short, the initial environment of a user task should be correct. For example, a spoolin task accessing a card reader containing information of U_i should only have access to a drum (disk) file if it belongs to U_i. In addition, any rights retrieved by the task initiator (such as for a program or data file) should in fact belong to the user. This leads to our second principle.

<u>Initiation</u>: If task T is initiated on behalf of U_i, then T has no access to external or stored user information $D_{j,k}$ unless $j = i$.

Verification of this principle should be fairly easy because it only entails looking at the domain of the initiated task (i.e. his capabilities) and the implementation of user data.

The final critical interface operation is the translation of user source programs. Here our concern is twofold: first, the translator should not give away a copy of the translated program and, second, he must correctly translate any rights in the program. This problem has not, to our knowledge, been addressed but it seems crucial to security. For if a translated program contains an instruction to, say, grant access to a file and the user did not specify that he wanted access granted, then his security has been violated. We summarize this principle as follows:

<u>Program Translation</u>: Suppose translator T is given a source program S_i of user U_i and produces object program O_i. Then no U_k, $k \neq i$, has a right (capability) to access O_i and all rights and right manipulations in S_i have been correctly translated into capabilities and protection primitives by T.

Adherence to this principle does not require that T be completely correct, only that it translate rights correctly. Taken together, the identification, initiation, and translation principles should insure that the interface processes do not violate security.

KERNEL PRINCIPLES

Many protection mechanisms with varying degrees of "robustness" or "completeness" have been proposed [2,6,9,18,20,21,22,23]. Regardless of the level of sophistication, however, we feel that three principles must minimally be adhered to if the kernel is to be adequate for security. First, the mechanism must be logically correct in that every action by every computation, whether user or operating system, is enforced and all changes to the protection state used by the enforcement mechanism are controlled.

<u>Enforcement</u>: An action α on object O by actor A is allowed if and only if there exists a capability (O,α) in the domain of A when α is attempted.

<u>Integrity</u>: Execution of a kernel protection primitive is the only way in which the protection state changes.

Subject to verification of the implementation, several existing protection mechanisms satisfy both of these principles [2,21,23].

Another basic property which a kernel should have is that no actor can give another more capabilities than he himself has. In addition, it has been recognized that explicit authorization should be required to Grant (copy) a capability [1,2,8,9,11, 21,23]. Without these properties, the kernel would hardly provide tools for controlled information sharing. We summarize this property by the following principle:

<u>Propogation</u>: Suppose actor A has a capability (O,(a,c)) where O is an object, a is the access A has to O and c is the control A has over O. Then A can grant (copy) a capability (O,(a',c')) to another actor only if:

(1) a' is a subset of a and c' is a subset of c, and

(2) "grant" ("copy") control is one of the attributes in c.

As with integrity and enforcement, many mechanisms satisfy this principle [2,21,23].

SERVICE PRINCIPLES

Having examined security requirements for interface processes and the protection kernel, it remains to have some way of insuring that user-invoked and background services are secure. For this purpose, confinement of user invoked services and isolation of background services appears to be sufficient. A <u>service</u> is a set of connected actors (e.g. nested procedures or communicating processes) which performs a computation on the parameters passed by its <u>customer</u>. The file access procedures (or processes) in a file system are one example of a service; another is the page swapping procedures invoked on a page fault in a paged system. An actor in a service is <u>engaged</u> by the customer if he was called (sent a message) either by the customer or by an actor engaged by the customer.

When an interface process requests operating system services on behalf of a user or when the user himself invokes services, it is important that the actors engaged by the user are confined. A service is totally confined if it is necessary to have authorizing capabilities in the parameters passed by the customer for any engaged actor to transmit information to another actor not in the service [1,3]. A confined service cannot therefore give information of one user to another. For the security of user invoked services, we want the following principle to hold.

Confinement: Any operating system service invoked by a user or on his behalf by a task initiator is confined.

This problem has been examined elsewhere [1,3,5,12] and a workable solution has been proposed which determines if a service is confined by examining its capabilities [3]. Since a large proportion of any OS is user-invoked services, this technique should greatly help in certifying security.

The final principle we advocate relates to background services performing system wide functions such as performance measurement.

Isolation: Every actor in an OS who is not engaged by an actor associated with a user is isolated from the information of every user.

Isolation has also been examined elsewhere [1,8,9] and requires that an actor not be able to examine or alter memory containing user information. An isolated actor can, however, be allowed to move information about or keep reference counts provided he does not change any capabilities or object mappings in an insecure way; two kernel primitives for manipulating memory and mappings without destroying isolation have been proposed [1]. While we advocate isolating background services from user information, it is not necessary to do so for security. What is required is to prove that these services do not divulge any information they examine; this condition is harder to prove than isolation, however.

CONCLUSION

We have attempted to present a workable definition of security and an approach to its verification. Some of the proposed principles have been examined and provable conditions have resulted. The approach of partitioning an operating system into small parts which can either be proven to be correct or at least can be proven to be secure without examining their code, in our opinion, holds the key to constructing a truly secure system. Much work remains to be done, however, to bring this hope to fruition. In particular, the concepts presented here need to be formalized and applied.

BIBLIOGRAPHY

1. Andrews, G.R., COPS - A protection mechanism for computer systems. Tech. Report 74-07-12 (Ph.D. Thesis), Computer Science Group, University of Washington, 1974.

2. Andrews, G.R., COPS - A mechanism for computer protection. Proc. Int. Workshop on Protection, IRIA, Rocquencourt, France, August 1974, 5-25.

3. Andrews, G.R., Concepts and conditions for confinement. Tech. Report, Dept. of Computer Science, Cornell University, 1975. In preparation.

4. Burke, E.L., Synthesis of a software security system. Proc. ACM 1974 Annual Conference, San Diego, 648-658.

5. Denning, D.E., Denning, P.J. and Graham, G.S., Selectively confined subsystems. Proc. Int. Workshop on Protection, IRIA, Rocquencourt, France, August 1974, 55-61.

6. Dennis, J.B. and Van Horn, E.C., Programming semantics for multiprogrammed computations. Comm. ACM 9, 3 (March 1966), 143-155.

7. Evans, A., Kantrowitz, W. and Weiss, E., A user authentication scheme not requiring secrecy in the computer. Comm. ACM 17, 8(August 1974), 437-442.

8. Graham, G.S. and Denning, P.J., Protection - principles and practice. AFIPS Conf Proc. 40 (1972 SJCC), 417-429.

9. Jones, A.K., Protection in programmed systems. Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, June 1973.

10. Jones, A.K. and Wulf, W.A., Towards the design of secure systems. Proc. Int. Conf. on Protection, IRIA, Rocquencourt, France, August 1974, 121-135.

11. Lampson, B.W., Protection. Proc. Fifth Princeton Conf. on Info. Sciences and Systems (March 1971), 437-443. Reprinted in Operating Systems Review 8, 1(Jan. 1974), 18-24.

12. Lampson, B.W., A note on the confinement problem. Comm. ACM 16, 10(Oct. 1973), 613-615.

13. Lipner, S.B. et. al., A panel session - security kernels. AFIPS Conf. Proc. 43 (1974 NCC), 973-980.

14. Neumann, P.G. et. al., On the design of a provably secure operating system. Proc. Int. Workshop on Protection, IRIA, Rocquencourt, France, August 1974, 161-175.

15. Popek, G.J. and Kline, C.S., Verifiable secure operating system software. AFIPS Conf. Proc. 43 (1974 NCC), 145-151.

16. Popek, G.J. and Kline, C.S., The design of a verified protection system. Proc. Int. Workshop on Protection, IRIA, Rocquencourt, France, August 1974, 183-196.

17. Purdy, G.B., A high security log-in procedure. Comm. ACM 17, 8(August 1974), 442-445.

18. Saltzer, J.H., Protection and the control of information sharing in Multics. Comm. ACM 17, 7(July 1974), 388-402.

19. Saltzer, J.H., Ongoing research and development on information protection. Operating Systems Review 8, 3 (July 1974), 8-24.

20. Schroeder, M.D. and Saltzer, J.H., A hardware architecture for implementing protection rings. Comm. ACM. 15, 3 (March 1972), 157-170.

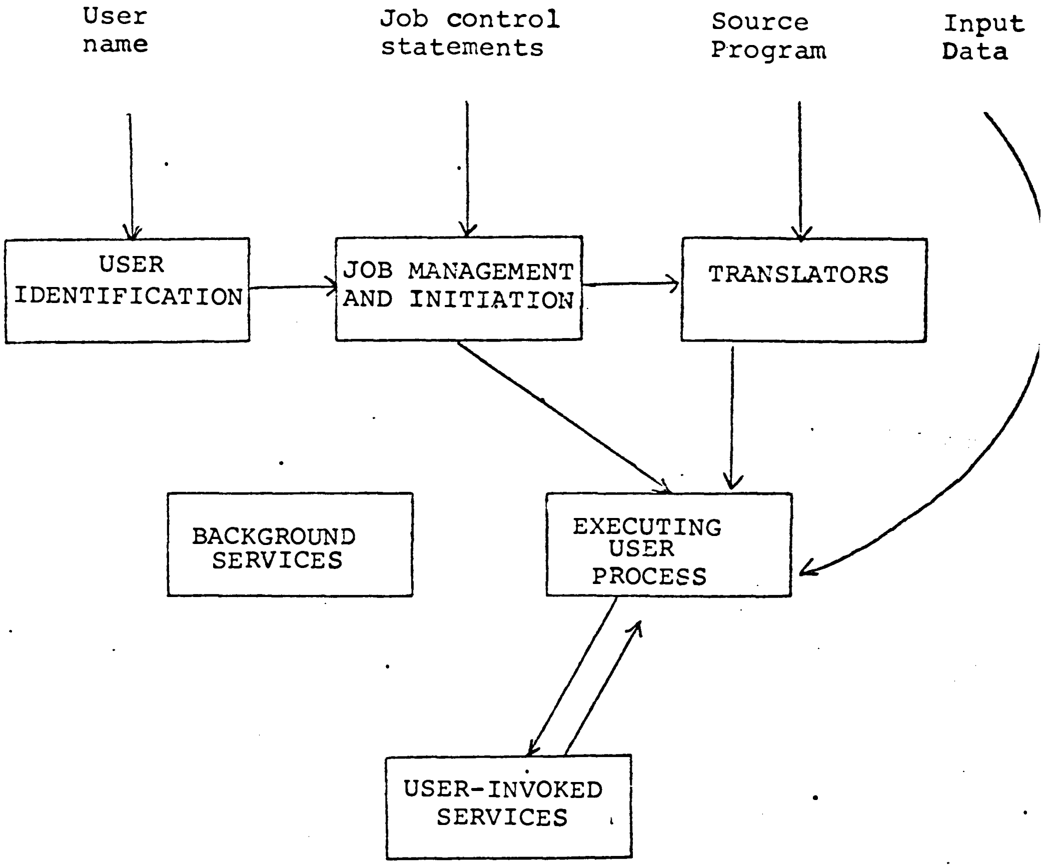
21. Spier, M.J., A model implementation for protective domains. Int. Journal of Computer and Infor. Sciences 2, 3 (Sept. 1973), 201-229.

22. Spier, M.J., Hastings, T.N. and Cutler, D.N., An experimental implementation of the kernel/domain architecture. Proc. 4th Symposium on Operating System Principles, Yorktown Heights, N.Y., October 1973. Reprinted in Operating Systems Review 7, 4(October 1973), 8-21.

23. Wulf, W.A. et. al., Hydra: the kernel of a multiprocessor operating system. Comm. ACM 17, 6(June 1974), 337-345.

Figure 1

OPERATING SYSTEM PARTITIONS AND FLOW



All of these partitions are implemented by using the SECURITY KERNEL.