ELECTRONIC SYSTEMS DIVISION

AIR FORCE SYSTEMS COMMAND

L.G. HANSCOM FIELD BEDFORD, MASSACHUSETTS

MCI - 73 - 1

January 1

JAMES P. ANDERSON
BOX 42
FORT WASHINGTON, PA. 19034

PRELIMINARY NOTES ON THE DESIGN OF SECURE MILITARY COMPUTER SYSTEMS

ROGER R. SCHELL, MAJOR, USAF

PETER J. DOWNEY, 1st LT, USAF

GERALD J. POPEK, 2nd LT, USAF

DIRECTORATE OF INFORMATION SYSTEMS TECHNOLOGY

DEPUTY
FOR COMMAND AND MANAGEMENT SYSTEMS

PRELIMINARY NOTES

ON

THE DESIGN OF SECURE MILITARY COMPUTER SYSTEMS

ROGER R. SCHELL, Major, USAF
PETER J. DOWNEY, 1LT, USAF
GERALD J. POPEK, 2LT, USAF

FOREWORD

This document is a collection of internal working notes produced by members of the Computer Security Branch, Directorate of Information Systems Technology, Deputy for Command and Management Systems, during the period of August - November 1972.

Although the preliminary nature of these notes is emphasized, we hope they will be an aid to understanding the direction of ongoing computer security efforts, until such time as more complete results are available. Three efforts now underway have been influenced by the ideas expressed here, and future products can be anticipated:

a. ESD-TR-73-51, Computer Security Technology Planning Study, by James P. Anderson, dated October 1972.

b. MITRE-MTR-2547, "Secure Computer Systems: Mathematical Foundations", by D. E. Bell and L. J. LaPadula.

c. Final report from Case Western Reserve University under the ESD(MCI) Statement of Work, "Abstract Model for Secure Computer Systems".

REVIEW AND APPROVAL

Publication of this report does not constitute Air Force approval of the report's finding or conclusions. It is published only for the exchange and stimulation of ideas.

[illegible]

MELVIN B. EMMONS, Colonel, USAF
Director, Information Systems Technology
Deputy for Command & Management Systems

TABLE OF CONTENTS

NOTES ON AN APPROACH FOR DESIGN OF SECURE MILITARY ADP SYSTEMS

Introduction

The military has a heavy responsibility for protection of information in its shared computer systems. The military must insure the security of its computer systems _before_ they are put into operational use. That is, the security must be "certified", since once military information is lost it is irretrievable and there are no legal remedies for redress.

Most contemporary shared computer systems are not secure because security was not a mandantory requirement of the initial hardware and software design. The military has reasonably effective physical, communication, and personnel security, so that the nub of our computer security problem is the information access controls in the operating system and supporting hardware. We primarily need an effective means for enforcing very simple protection relationships, (e.g., user clearance level must be greater than or equal to the classification level of accessed information); however, we do not require solutions to some of the more complex protection problems such as mutually suspicious processes.

Based on the work of people like Butler Lampson we have espoused three design principles as a basis for adequate security controls:

a. _Complete Mediation_ -- The system must provide _complete mediation_ of information references, i.e., must interpose itself between any reference to sensitive data and accession of that data. All references must be validated by those portions of the system hardware and software responsible for security.

b. _Isolation_ -- These validation operators, a "security kernel", must be an isolated, tamper-proof component of the system. This kernel must provide a unique, protected identity for each user who generates references, and must protect the reference-validating algorithms.

c. Simplicity -- The security kernel must be simple enough for effective certification. The demonstrably complete logical design should be implemented as a small set of simple primitive operations and system data base structures that can be shown to be correct.

These three principles are central to the understanding of the deficiencies of present systems and provide a basis for critical examination of protection mechanisms and a method for insuring a system is secure. It is our firm belief that by applying these principles we can have secure shared systems in the next few years.

Deficiencies of Present Systems

Most current computer systems exhibit a complex, ad hoc security design with diffuse implementation that violates our third principle of simplicity. Large portions of complex operating systems execute in an all-powerful supervisor state, so that the entire operating system has potential security implications. Whatever nominal security controls exist in such bug-prone monoliths are not effectively isolated (in violation of our isolation principle) and so can be tampered with through errors or trap doors in other parts of the operating system.

The significance of these inherent security weakness has been amply and repeatedly demonstrated by the ease with which contemporary systems (such as OS/360 and GCOS) have been penetrated. Unfortunately, this lack of an underlying design methodology cannot be effectively overcome by ad hoc "fixes" and "security features" built on an uncertain foundation.

Certification

A naive (but occasionally attempted) approach to insuring the security of a complex operating system is to have a penetration team of "experts" test the system. It is supposed that repeatedly unsuccessful penetration attempts demonstrate the absense of security "holes". Such a test approach is primarily limited to penetration attacks in areas indicated by the particular background and experience of the individuals involved. A security evaluation through such attempts may reveal weaknesses of a system but provide no indication of the presence or absence of trap doors or errors in areas unnoticed by the attack team. The failure of an attack team to notice a particular penetration route does not prove or certify that an actual penetration attempt will overlook it at a later date. The underlying concern is that an active hostile penetrator is not particularly thwarted by the various flaws found and fixed through testing so long as there remains _just one_ vulnerability that he can find and effectively exploit.

On the other hand our three principles lead to a simple, well-defined subset of the system totally responsible for information protection. We expect that the primitive functions of this small, simple kernel can be tested by enumeration, and other parts of the system are not relevant to security. As a result most system changes will not affect the kernel, so routine system maintenance will not require repeated recertification.

Practical Mechanisms

An abstract security model is needed in order to evaluate the adequacy of protection mechanisms. Lampson's capability (i.e., access matrix) model has proven a useful departure point, and we have applied two design techniques for developing a specific secure design:

a. The model is represented in various levels of abstraction. The design process transforms an initial abstract model of all the system's protection relationships (derived directly from the system's specific definitions of security, thus leading to a model that is secure by hypothesis) into subsidiary levels of abstraction. As the design progresses from level to level the representations of the model become more specific and culminate in specific hardware features. The inter-level transformations, chosen for reasons of efficiency as well as utility, can ultimately be implemented as primitive operations of the kernel, and since the inter-level transformations preserve the initial protection relationships, we can prove that the resulting design is secure.

b. The kernel design is simplified by including only those relevant operations that modify access control data bases, but not those that merely read this control information that is not itself being protected against disclosure. Consider as an example a demand paging system. At some level of abstraction page table entries represent capabilities that must be carefully controlled, so the kernel will have a primitive for changing page table entries; however, the page replacement selection algorithm should not be in the security kernel.

Using this model, descriptor-based addressing available in advanced processor hardware is seen to offer a most promising basis for a security kernel design. In terms of our first design principle (complete mediation), this addressing hardware validates each memory reference by a user's process: it interprets the required access, specified in the applicable descriptor. The security kernel insures security through its primitive operations, which are invoked by the remainder of the operating system to maintain the descriptors. Because access control is vested in the well-defined and bounded descriptor mechanism, kernel software functions are few enough and simple enough to make certification tractable, as required by simplicity, our third design principle.

Descriptor-based isolation mechanisms (such as Schroeder's hardware implemented rings for Multics) can provide effective as well as efficient protection of the security kernel. Thus, as implied by our second design principle (isolation), an antagonist could have complete freedom within the remainder of the system without compromising the protection provided.

Prospect for the Future

In the Air force we are pursuing a development effort for providing secure shared systems in the next few years. In cooperation with the MITRE Corporation, we are already applying our three design principles to shared communications processors in the laboratory, and we have begun to extend these ideas to a design for a shared, general purpose computer system.

We are confident that from the standpoint of technology there is a good chance for secure shared systems in the next few years. However, from a practical standpoint the security problem will remain as long as manufacturers remain committed to current system architectures, produced without a firm requirement for security. As long as there is support for ad hoc fixes and security packages for these inadequate designs, and as long as the illusory results of penetration teams are accepted as a demonstration of computer system security, proper security will not be a reality.

ON THE DESIGN OF SECURE SYSTEMS

SECTION 1 PHILOSOPHY

Our intent is to provide a basis for the design of multiuser computer systems in which there exist security mechanisms that provide: 1) a useful degree of flexible security and 2) a high degree of confidence in the integrity of the mechanisms.

The problem of computer security is well recognized and a number of systems and system designs have been proposed. However, it is often difficult to evaluate these efforts without understanding the assumptions implicit in the system design or recognizing what portion of the security problem the system purports to solve.

Hence, we briefly state in general terms our conception of that part of the current military computer security problem that we will consider, and later restate this general conception more exactly. The kind of security that is currently desired is not complex in its functional capability. We do not demand the ability to handle the problems of aggregation, inference, or mutually suspicious subsystems. We do not attack those problems which seem to require a monitoring and general understanding of the _use_ to which information will be put, excepting rather simplistic controls like read, write and execute, and hence are satisfied by a set of simple decision rules which operate on information recorded in the system, not unlike the class of facilities that a number of timesharing systems provide today.

The critical requirement is extremely high _integrity_: great confidence that the specified design of the security facilities of the system are in fact guaranteed. We recognize, of course, that the system must provide useful capabilities, since otherwise a guaranteed design or implementation is vacuous. That is, the proposed security controls must allow the implementation of a multiuser computer system with functional capabilities not unlike a number of today's common commercially available time sharing systems.

In an attempt to fulfill these goals, the following strategy is proposed: develop a simple logical design whose correctness can be verified, and whose elements are both simple enough and close enough to real system features so that implementation of the model is reasonably straightforward.

As the abstract model is developed, we shall be guided by the idea of a <u>kernel</u>. We intend to isolate that portion of the system responsible for security and place it in a protected part of the system, in a manner analogous to the way in which current supervisors are segregated from user programs. It will be necessary to demonstrate that this segregation is performed in a way that guarantees the kernel's integrity and also guarantees that the kernel is always invoked to arbitrate attempted references. These tasks are eased by the fact that we will design our security model so that it can aid in protecting itself.

By segregating the responsibility for security, the problem of verifying the system's security mechanisms becomes that of: 1) demonstrating that the kernel is always invoked, and 2) verifying that the kernel operates properly. The problem has been greatly reduced from that of verifying properties of an entire operating system to that of verifying a (presumably) small portion of it.

The design model should consist of several levels of abstraction. The top level is a logical description of security systems; the lowest level closer to a possible machine implementation. Higher levels are more machine independent than lower levels. The intent is to prove the correctness of an upper level machine independent model, and demonstrate that translations to lower, more specific levels preserve the relevant properties of the top level. Through the use of this top down informal structure, we hope to demonstrate the correctness of an implementable design for a secure system. Lest readers labor under any misconceptions, it should be pointed out that while the "proof" structure is top down, the system design certainly is not. Fairly well defined ideas of the desired end product exist. The top down approach is primarily for purposes of description and proof.

A remark should be made concerning the meaning of "correctness", and "proofs of correctness". A system cannot, in a vacuum, be proved correct. It may, however, be possible to demonstrate that a system design agrees with, or fulfills, certain _external_ criteria, that is, conditions which are not explicitly part of the design. These external criteria specifically characterize that "computer security problem" which we consider.

We will demonstrate that in certain cases these explicit, external criteria can be made part of the system design, in such a manner that they are always applied, reducing the problem of an informal correctness proof.

A last constraint is placed on the design by the need for efficiency. The security mechanisms should not markedly degrade the price/performance characteristics of a system. The effect of this constraint is more apparent as discussion moves closer to implementation.

SECTION 2 - A SECURITY META-MODEL

Introduction

The following approach is intended as a guide for the logical design of computer security systems. The description applies to a wide class of security systems, including most of those in practice or proposed today.

Naturally, then, the meta-model does not provide an instance, or implementation, of a useful secure system. Using the meta-model, for example, one can provide inappropriate standards for correctness, or one can design a system that is not useful. As a case in point, whether or not provision is made for the operation of "cooperating, mutually suspicious process", is irrelevant to the meta-model.

However, the security meta-model allows one to relate various specific models, and provides a specific guide to those actions necessary to guarantee the correctness of a security design.

Notation

In the following discussion, some non-standard notation is used to linearize formats. Several conventions should be pointed out. Subscripts are enclosed in square brackets. Sets are labelled by capital letters, and elements of that set are generally labelled by the same letter, but in lower case and subscripted. Hence a[j] refers to the j-th element of the set A.

It is occasionally necessary to speak of the names of members of a set, rather than the members themselves. The set of names which corresponds to a set of elements is denoted by an underline. So, for example, the set $\underline{A}$, with elements $\underline{a}$[j] is a set whose elements are names, corresponding to the set A.

Last, the power set of a set X is written P(X).

Brief Description

The model is described in set theoretic language, and has six major components. First is the set O of security objects: the elements of the model, reflecting those physical or logical parts of a computer system that need to be controlled, protected, or whose status needs to be guaranteed. The objects are partitioned into disjoint classes, each containing objects of similar characteristics. An incomplete list of examples includes terminals, communication lines, processes and files.

Second, a set A of access types is presented. Each access type is a program which effects a particular variety of access, such as read, write, or execute. An attempted access operation is then completely specified by an access type and some meaningful collection of objects, i.e. a particular process being directed from a given terminal attempting to reference a specified page in memory.

Third, a collection of descriptive data D[k], from the set of all possible descriptive data collections D is required. D[k] specifies the information that forms the basis by which security decisions will be made. The subscript k indicates a time dependency.

Fourth, an evaluation program, E decides, for any meaningful grouping of objects, what operators are to be allowed.

Fifth, an update program U is characterized separately. This program is the means by which the descriptive data are changed. Operationally, this is the manner by which access decisions may be altered.

In many real implementations, the distinction between the evaluation program and update program may not be clearcut, since the descriptive data is likely to be stored and protected like any other security object. Both programs are treated here so that their similar nature is apparent. Nevertheless, the distinction will be useful since implementations of the two programs may differ. E, while likely to be software implemented, calls upon access programs to do its actual work, and these may be at least partly if not wholly built in hardware. U on the other hand in many cases will be almost exclusively software and actually changes the formatted descriptive data.

Last, *external correctness criteria* are required. These are a set of rules, or standards T, by which the system is to be adjudged correct. These standards must be external to the system description up to this point in order to be meaningful.

A security system S is then specified by the six-tuple:

S = (O, A, D, Ɇ, Ø, T).

The Components of the Model

Security Objects

The first component of the model, the security objects, is a finite set O:

O = {o[1], o[2], ..., o[z]}.

These are the *only* objects to which access will be controlled by the model, and by a resulting implementation.

Access Types

The second component of the model is a set of access types:

A = { a[o], a[1], a[2], ..., a[w] }

Each a[i] is a program whose effect will be to provide a particular variety of access, read, write, or execute for example. The list of arguments for each a[i] must be finite and contain names of security objects. In addition, a[o] is designated as the *null* access program. This program will be invoked when access is to be denied. It can keep audit trails, set up warnings to administrators, etc.

Descriptive Data

The third component, the descriptive data, is merely a set of tuples:

$$D[k] = \{ d[k,1], d[k,2], \ldots, d[k,v] \},$$

with some finite upper bound set on v. We depart somewhat from our strict set theoretic notation by speaking of the structure of a tuple.

Each tuple is only assumed to have a bounded number of entries, the first of which acts as a "data descriptor" to distinguish among tuples of different formats and content.

For example, one type of tuple might be an encoding of a matrix entry in Lampson's model [4]; the entry expressing an access relation between two security objects. Another might express a property: user x belongs to project y, or has clearance z. A property may also be valid only for several users jointly. Such circumstances do not fit naturally into a matrix representation of the descriptive data, so tuples are preferred here.

Explicit use of the structure of the descriptive data will not be made in the following discussion of correctness, although it is necessary in the more detailed proof. The finiteness of both the length and number of tuples will be useful here, however.

Let X^* be the set of all allowed tuples, and $D = P(X^*)$ the power set of X^*. Then $D[k]$ is some member of $P(X^*)$.

Evaluation Program

The third portion of the model is an evaluation program E which uses descriptive data to make decisions concerning access. For any evaluation program, the list of arguments is composed of some fixed number of objects from each partition of the security objects O, and an access type; the name of an element in A. For convenience, those objects are denoted by θ.

The task of the evaluation program is to decide whether or not the specified objects may be associated in the manner expressed by the access type and to indicate an appropriate action. That indication is done by selecting the appropriate access program and specifying its proper arguments.

The evaluation program Ɇ takes a list of object names, a particular descriptive data configuration, and the *name* of an access type (names of elements are underlined); and returns the allowed access *program* together with the argument list for that access program.

Ɇ is composed from an access rule E. E is a fairly arbitrary program that is assumed only to 1) terminate, returning *true* or *false*, and 2) be read only.

The intent is that E describe conditions to be fulfilled in order to allow access. It may be an arbitrary function of its arguments, although often such programs are fairly simple.

Then the program Ɇ may be written as follows:

```
Ɇ : proc (θ, D[k], a[j]) returns list;
lock;
if E(θ, D[k], a[j])
then begin unlock; call a[j](θ) end;
else begin unlock; call a[o](θ) end;
end;
```

The list which is returned specifies an access type and the argument list for that program. The arguments for E are the same as for Ɇ itself.

The functions *lock* and *unlock* are understood to act on a single semaphore, as Dijkstra's operators P(x), V(x). It is necessary to coordinate the operation of Ɇ and Ʉ so that Ɇ is not reading D[k] while Ʉ is updating D[k]. Otherwise, it would not be possible to prove that Ɇ and Ʉ perform in all cases as claimed.

Update Program

The update program is the means by which descriptive data is changed. Hence it is the manner by which decisions that the evaluate program makes can be affected. Let θ' denote the set of arguments for the update program which are security objects, D[y] is the current descriptive data, and D[z] is the data to which it is desired to change. Ø yields either the original data, prohibiting the change, or the new data, having allowed the change.

The update program, too, is composed from some effective procedure U, similar in purpose to E, and so the update program Ø may be written as:

```
Ø : proc (θ', D[y], D[z]) returns element of D;
lock;
if U(θ', D[y], D[z])
then begin unlock; return D[z] end
else begin unlock; return D[y] end
end;
```

The arguments for U are the same as for the procedure itself.

The Correctness Criteria

The security objectives of the access control system are the qualities that it is necessary to guarantee. For a certain well defined class of criteria, there is a straightforward method of taking a logical description of a security system and altering that model to provide a derived system model in which the given correctness criteria hold.

The correctness criteria are expressed as a set T of predicates:

$$T = \{t[1], t[2], \ldots, t[q]\}.$$

These are the predicates that must be proven true for the system.

In this model, predicates may be expressed in one of two forms, and so T is partitioned into two subsets T1 and T2 corresponding to the two alternatives.

If t[i] is in T1 then it may be any predicate expressible in the following functional form:

$$t[i] : \theta \times D \times \underline{A} \rightarrow \{\underline{true}, \underline{false}\}.$$

The interpretation of predicates in T1 is that the object list from θ may be associated with access type a[j] in A and a given D[k] in D _only if t[i] is true_.

If t[i] is in T2, then it may be any predicate expressible in the following functional form:

$$t[i] : \theta' \times D[j] \times D[k] \rightarrow \{\underline{true}, \underline{false}\}$$

The interpretation is that the descriptive data D[j] may be changed to D[k] by the objects expressed by θ' only if t[i] is true.

Let

$$\mathcal{T}1 = \text{And } (t[i]) \text{ for all } t[i] \text{ in } T1 \text{ and}$$

let

$$\mathcal{T}2 = \text{And } (t[j]) \text{ for all } t[j] \text{ in } T2.$$

$\mathcal{T}1$ and $\mathcal{T}2$ take the same arguments as the t[i] and t[j], respectively.

To demonstrate that a system is correct, it is necessary to guarantee the truth of $\mathcal{T}1$ and $\mathcal{T}2$. Below, a simple way is shown to take any security system S and derive from it a system S' for which the given $\mathcal{T}1$ and $\mathcal{T}2$ are <u>true</u>.

<u>Derivation of Correct System</u>

<u>System Specification</u>

As described, a security system S is a tuple:

$$S = (O, A, D[o], \mathcal{E}, \mathcal{U}, T)$$

O is the object set, A is the set of access types, D[o] is taken as the set of tuples which comprise the initial descriptive data, $\mathcal{E}$ is the evaluation program, $\mathcal{U}$ is the update program, and T is the set of predicates to be guaranteed.

For a particular system S, the entries A, $\mathcal{E}$, $\mathcal{U}$, and T are fixed. The descriptive data D[k] may be varied by use of $\mathcal{U}$. Then the <u>state</u> of a security system S can be completely expressed by its descriptive data D[k], for some k. The update program is the means by which a system S may change states and the compound predicate $\mathcal{T}2$ expresses the constraints on allowed state changes. The evaluation program $\mathcal{E}$ "interprets" a particular state, and $\mathcal{T}1$ expresses the constraints on $\mathcal{E}$.

Given a security system S = (O, A, D[o], $\mathcal{E}$, $\mathcal{U}$, T), system S' = (O, A, D[o], $\mathcal{E}'$, $\mathcal{U}'$, T) is produced by the following *inclusion step*.

$\mathcal{E}'$ is derived from $\mathcal{E}$ by the following change. Replace "E(...)" by "E(...) *and* $\mathcal{T}1$(θ, D, a[j])".

$\mathcal{U}'$ is derived from $\mathcal{U}$ by the following change: Replace "U(...)" by "U(...) *and* $\mathcal{T}2$(θ', D[y], D[z])".

Correctness Proof

First it is helpful to define a few terms.

A state D[n] of a system

S = (O, A, D[o], $\mathcal{E}$, $\mathcal{U}$, T)

is *valid* if and only if D[n] can be obtained from D[o] by a finite number of applications of $\mathcal{U}$ and, for each such transition from state D[k] to D[k+1],

T2(θ', D[k], D[k+1]) = *true*

for some θ'.

Second, a state D[k] is *accurately interpreted* if and only if for any θ and any j:

$\mathcal{U}$(θ', D[k], D[j]) = (θ', a[o]) whenever $\mathcal{T}1$(θ, D[k], a[j]) = *false* (where a[o] is the null access type).

Then to say that a system S is *correct* is meant the following:

1) Every state obtainable from D[o] is valid, and

2) Every valid state is accurately interpreted.

We now state the following (system correctness) *theorem*;
Given a security system

S = (O, A, D[o], Ɇ', Ø', T) with T partitioned

into T1 and T2;

and S' = (O, A, D[o], Ɇ', Ø', T) derived from S

by the *inclusion step*

then S' is correct.

Proof Sketch

An easy way to prove the theorem is by contradiction. Suppose the theorem false. Then, by definition of *correct*, S' reaches an invalid state, or a valid state is inaccurately interpreted.

Case 1: Assume an invalid state. Label that invalid state D[k]. Then there must exist a sequence of states D[o], D[1], D[2], ..., D[k] such that Ø (θ[i], D[i], D[i+1]) = D[i+1] for all i<k, since Ø makes the transition from state to state.

Now D[o] is valid by definition. D[k] is invalid by assumption. Then there must exist a non-negative interger j, less than k, such that D[j] is valid and D[j+1]) is invalid. Hence, by definition of valid, 𝒯2(θ, D[j], D[j+1]) is *false*. But Ø(θ, D[j], D[j+1]) = D[j+1]. By inspection of Ø, these two conditions cannot hold, and hence a contradiction is reached.

Case 2: Assume an inaccurately interpreted valid state. Call that valid state D[k]. Then by definition of an accurate interpretation, for some θ[i] and a[j], the following is true.

𝒯1(θ[i], D[k], a[j]) = *false* and

Ɇ (θ[i], D[k], a[j]) ≠ (θ, a[o])

By inspection of $\mathcal{E}$, this is a contradiction. Hence every valid state is accurately interpreted.

Both cases are impossible. Hence the theorem cannot be false.

qed

This proof is of course nearly tautologic in nature.

Discussion

This security meta-model and the inclusion technique are intended as an aid in the design of secure multiuser computer systems. Hence some of the assumptions and implications inherent in the choice of language, model, and technique ought be made explicit.

The primary influence in this meta model was the realization that its value is solely in its ability to aid the implementation of a demonstrably secure system. Hence the model and its elements must conform to the modules and capabilities of computer systems not unlike those in existence today. At the same time, a simplicity and coherence was desired, reasonably free of implementation questions, that would provide some understanding of the contemporary security problem. It is felt that the basic concepts explicated here are a reasonable start toward these goals, although it is freely admitted that exposition, notation and other details may require improvement.

A number of implementation implications of this meta model can be mentioned.

First, it should be pointed out that effective procedures exist for the update and evaluation programs, the predicates from which they are composed, and the predicates which make up the correctness criteria. This fact is a result of the finiteness of all the sets involved in the meta-model. That effective procedures exist for all the predicates in the theorem set T makes the inclusion technique actually useful. In certain actual implementations of course, it may be possible to demonstrate the truth of some of the correctness criteria without dynamically verifying them at run time.

No claim of efficiency is made in this model, since for any particular system the predicates may be complex and the descriptive data specified in a manner that requires a great deal of work to check the given predicates. On the other hand, as will be demonstrated in a companion paper, the correctness criteria predicates for certain real problems are rather simple, and careful design of the descriptive data can greatly aid efficiency while remaining faithful to this meta-model. It is this fact which really guarantees the effectiveness of the inclusion step.

The next abstract level is sketched in a companion paper in order to demonstrate that a useful security system can be described with the language of the meta-model, showing that the meta-model is not vacuous.

It is intended that the kernel of a computer system include everything that this meta model contains, and nothing else. Hence the meta model defines the boundaries of the kernel, and the ability to use the kernel to protect parts of itself will allow one to provide carefully controlled access to the kernel itself.

Summary

The meta model provides a language for describing a useful class of security systems. It easily lends itself to the use of a technique which guarantees that the objectives of the system are fulfilled by the model. The concepts of the model are relatively simple and bear a reasonably close relation to the kinds of computer systems in existence today, suggesting the possibility of providing, with high confidence, a faithful implementation of the model. An accurate implementation of a desired security design is, after all, the primary goal of all of this work.

BIBLIOGRAPHY

1. Conway, R. W., et.al., "On the Implementation of Security Measures in Information Systems," Comm. ACM 15, 4 (April, 1972), pp. 211-220.

2. Elspas, Levitt, Waldinger, Waksman, "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys, Vol. 4, No. 2, June 1972, pp. 97-147.

3. Friedman, T., "The Authorization Problem in Shared Files," IBM Systems Journal 9, 4, 1970, pp. 258-280.

4. Graham, G.S. and Denning, P.J., "Protection - Principles and Practice," AFIPS Conf. Proc. 40, (SJCC 1972), pp. 417-429.

5. Graham, R.M., "Protection in an Information Processing Utility," Comm. ACM 11, 5 (May, 1968), pp. 365-369.

6. Hoffman, L., "Computers and Privacy: A Survey," Computing Surveys, Vol. 1, No. 2, June 1969, pp. 85-103.

7. Hoffman, L.F., "The Formulary Model for Flexible Privacy and Access Controls," AFIPS Conf. Proc.39,)FJCC 1971), pp. 587-601.

8. Lampson, B.W., "Dynamic protection structures," AFIPS Conf. Proc. 35. (FJCC 1969), pp. 27-38.

9. Lampson, B.W., "Protection," Proc. 5th Princeton Conf. on Information Sciences and Systems, (March, 1971), pp. 437-443.

MUSINGS CONCERNING A SPECIFIC SECURITY MODEL

(The following thoughts were sketched under significant time constraints and are released in their present form only with considerable reluctence. Nevertheless, it is hoped that a useful partial explication is provided of the applicability of the ideas previously presented, specifically the kernel and the meta model design approach.)

With the general outline of the security meta-model in mind, we sketch a model of a particular security system. It is not an extremely general one, but rather is intended as a statement of current military needs in a context that both provides a basis for a proof of correctness and can lead fairly directly to an implementation. To make it clear that implementation is possible, the flavor of the structure is taken from the existing file system of Multics.

A few notes should be made concerning the intended environment of this model. An on line multiuser computer installation is expected, where the mechanisms proposed in this model, directly or indirectly, check every reference made to information contained in the system.

The Objects

The object set O might be partitioned into four subsets:

Ot = a set of terminals

Ou = set of users

Od = set of data objects

Os = set of security objects

Terminals are meant to be representative of the entire class of I/O devices, and could include teletypes, printers, tape drives and the like. For every user recognized by the system, there is an object in Ou; the "user process". Data objects include both executable and non-executable objects; the items that the system is intended to protect. Lastly there are security objects. Security objects contain the information upon which security decisions will be made. There are two distinctions between data objects and security objects. First, security objects will have a rigidly enforced internal structure necessary for proper operation of the security system, while data objects are format free - completely free form internally. Second, security objects will be accessed directly only by the decision and update programs.

Names of objects will be required distinct, of course.

<u>The Descriptive Data</u>

As already mentioned, the descriptive data is contained in the security objects. This containment provides a manner by which access to the descriptive data itself can be controlled by the mechanisms of the model.

Any security object os(i) is an ordered list of descriptors

$$os(i) = \{d(1), d(2), \ldots d(n)\}$$

where a descriptor is an n-tuple, and $n \geq 5$

$$d(i) = [\underline{o}, \underline{m}, \underline{c}, \underline{p}, r(1), r(2), \ldots, r(k)]$$

In any descriptor, $\underline{o}$ is the name of a member of the object set O.

The second element, m is a member of the mode set M.

$$M = \{1, 2, 3, 4, 5, 6, 7\}$$

The compartment list is the third entry.

It is useful to be able to label an object as a member of any number of several areas, or compartments. Hence a set of compartments is defined:

$$c' = \{c(1), c(2), \ldots, c(17)\}$$

and for convenience we also define

$$c = P(c) \text{ , the power set of } c'.$$

Any compartment list is the name of an element in C.

The remaining entries except for P are relations; there will be an arbitrary number of them. To describe relations, define first the access type set A'.

$$A' = \{\text{copy, write, execute, read, update}\}$$

and also $A = P(A')$.

Then any relation is a 2-tuple [ou, a] whose first entry is a user name: of an object from Ou, and whose second entry is the name of an element of the set A

Each member of the access set can be thought of as a program whose effect is to provide a particular variety of access. The necessary parameters are specified later, but it is assumed in this section that these programs are correct. What such programs actually do, of course, provides the semantics for their names.

It is intended that the access types copy, write, and execute apply to data objects: write and execute have the usual interpretation, while copy is synonomous with the usual definition of read. Copy is a better mneumonic for the actual ability provided. Read and update are access types that refer to security objects, and will have the suspected meaning. The last entry in the descriptor not yet mentioned is p. This entry is a specification of the actual, machine dependent location of the object whose name is the first entry in the descriptor.

The _format_ of the security objects' internal structure has now been informally defined. Some additions will be required like indicators of the number of relations in a descriptor and the number of descriptors in a security object. Additional restrictions, on the actual _content_ of security objects, will be imposed by initial conditions and the updating procedures.

A Brief Discussion

Before continuing, it may help to discuss the motivation for the format selected, and the intended use to which the data will be put by the access and update programs. One will be able to represent the total descriptive data by a tree, where the nodes are security objects, and the edges from father to son are indicates by descriptors whose mode entry is s, for security object. A tree link lies between a security object named by the entry. Descriptors with other modes specify terminal leaves: the other objects, terminals, users, and data in the model. This tree structure provides the manner by which access to descriptive data can be controlled, since each node contains the information relevant to access control for each of its sons. Access to the root node is treated differently - it will be free for read, but not possible to change.

The update program will guarantee that the name actually stored in a descriptor is unique: no two descriptors will have the same name entry.

The totality of information about objects that the security system will employ to make access decisions is contained in the descriptor.

The Evaluate Program

The program is the manner by which the descriptive data is interpreted to control access to data objects.

First, we assume the existence and correctness of the programs which make up the access set.

Each such procedure takes as arguments a user name, a terminal name, and a data object name. Its action is to perform those hardware and/or software operations necessary for the access to take place.

In addition, we assume the existence of two correct procedures, user and terminal, which return the name of the user object which has initiated the current access request, and the terminal from which the request was initiated, respectively. In addition, we assume the existence of a access type program, that returns the kind of access requested: the name of an element in A; and a reference program that returns the name of the data object to be referenced. (These two programs need not be proven correct.) We also assume the existence of a program null which may be a nop, but may also initiate recording of certain parameters for later inspection. Null is only guaranteed not to grant any access.

The evaluate program in its initial state is relatively simple. To keep questions of implementation buried for the moment, we assume the existence of another correct program.

Relation (b, c, d) has arguments b=data name, c=user name, and d=access type name, the name of a program in A'. This program returns true iff: 1) there exists a descriptor entry specified by b, and 2) there is a relation tuple [c, k] in that descriptor, where k specifies a subset of the access types which includes d.

An initial evaluate program might then be written following the outline in the security meta model, but with relation (reference, user, access-type) replacing E in the evaluate program Ɇ. Note that while the terminal involved in this activity has not been included in the check, it would be a simple matter, given the existence of the routine terminal.

The Update Program

To more easily describe the update program at this level we again assume the existence of several programs:

create (o, os) creates the object with name o and adds a descriptor in os with default sensitivity and compartment list.

delete (o, os) destroys the object o and removes its descriptor from os.

read (os, i, j) returns the value of the j-th entry in the i-th descriptor in security object os.

write (_os_, i, j, val) sets the contents of the j-th position in the i-th descriptor in security object _os_ to val, if there exists an i-th descriptor.

We assume that the above programs are correct. We also assume that there is some mechanism, not required correct, by which a user program may communicate its wishes to the update program. The set of arguments with which the update program must be invoked are: 1) the name of the object whose descriptive data it is wished to change, 2) the name of the security object to which the object belongs, 3) the operation that is desired (which program to invoke), and 4) the relevant input parameters to that program (the desired new values in a descriptor).

It should be fairly straightforward to sketch an update program, given the outline in the meta model and the above correct routines.

Theorems

It is now necessary to specify the objectives of the system design. These, in some reasonably specific language, are the criteria to which it is desired the system conform. We first state the requirements, as currently understood, in rather informal English, and then begin to formalize them in terms of the specific model at hand. These requirements are relatively simple, and do not provide some of the guarantees that are currently desired by some segments of the computer community. However, at this point, it is believed that current and short range future military requirements would be satisfied. Informally, there are four requirements. 1) No user shall have any access to an object if the sensitivity rating of the user, at the time that initial access is attempted, is less than the sensitivity rating of the object. 2) No user shall have any access to an object if the set of compartments associated with the object at the time of initial attempted access is not contained by the set of compartments associated with the user. 3) No user shall have any access to an object unless authorized by a "need to know" specification at the time of initial attempted access.

The problem of demonstrating that these criteria are always applied in this model can be approached in the following way. First prove that the format, or structure of the data base will have the properties that are described in the descussion earlier. This amounts to proving a number of assertions about the effect of the update program.

Then state theorems one through three algorithmically. Modify the decision process in the access program to invoke the above algorithms as part of the decision process itself in such a manner that a) it is simple to show that the algorithms are always applied in the decision process, b) the parameters they are supplied are appropriate, and c) the result of the algorithms has a controlling effect on whether or not access is granted.

As an example, we restate the first two requirements below, using the notation: sensitivity (x) and compartment (x) to mean the sensitivity and compartment entry in the descriptor for object x, respectively.

```
proc label-check (user, object);
check <- true;
if sensitivity (object) > sensitivity (user)
        then check <- false;
if compartment (object) ¢ compartment (user)
        then check <- false;
return check;
end;
```

In the above, the symbol > means the binary arithmetic operator "greater than". The symbol ¢ is the negation of the set theoretic property of "contained in". It is presumably clear that programs for all of the operations and checks required for the procedure <u>label-check</u> are straightforward in light of the data base provided by the security objects.

The decision program is then modified by replacing "<u>relation</u> (...)" by

"<u>relation</u> (<u>reference</u>, <u>user</u>, <u>access type</u>) and <u>label check</u> (user, object)".

The truth of the three requirements can be guaranteed in this manner, if in addition a consistent data structure is assumed for the security objects.

This approach is equivalent to dynamic checks at run time of the state of the system. Certainly it is possible that careful construction of the logical structure of the system could obviate the need for some run time checks, in a fashion analagous to certain programming languages. While that approach might be more efficient, these checks do not appear particularly costly. Also, the logical correctness of the system probably could be more easily demonstrated under these circumstances, particularly in the face of changes to the system.

The preceding sketch has been intended _only_ as a resonability argument in support of the viability of the security meta model. There is no claim here of the accuracy of the detail. Rather, it is only argued that the highly modular, tree structured proof structure for a security kernel is a viable and effective manner to deal with the task of correct security system design.

SECURE MILITARY COMPUTING SYSTEMS

OUTLINE

CHAPTER 1 INTRODUCTION

1.1 Secure Operating Systems - General Goals

The security aspect of shared computer systems has in the past not received preeminent concern. Questions of efficiency and flexibility have forced it into the background of the design process.

The military, however, is faced with the spectre of irrevocable compromise of classified information, should a flaw exist in the system's security. A single cunning and malicious user may employ a bug to penetrate or degrade an essential system, which, once penetrated in secrecy may even be covertly and continuously tapped for intelligence.

As an example, Goheen and Fiske (4) report a successful penetration of an IBM System/360 operating in a classified environment. At the end of the study, the entire system was essentially open to the penetrators in complete secrecy.

In the past the military has insured protection of a sensitive computer facility by segregation of the equipment, limiting physical access to the equipment, and forcing on-site usage. Today the problem is to insure security in a modern time-shared multi-access, multiprogrammed system in which remote users with different clearance levels can run concurrently with data files and programs of varying clearance levels.

The military will adopt in the near future extraordinarily high standards for security certification of equipment and software. Some steps have recently been taken toward analysis of the military computer security problem, and toward articulating methodology for designing certifiable security systems (8), (9), (10).

Schell (10) has proposed three design principles for security mechanisms: _complete mediation_, _isolation_ and _simplicity_:

(1) The system must provide immediate and _complete mediation_ between reference to and retrieval of information, validating all such references using a special subsystem.

(2) This subsystem, the "security kernel", must thwart any attempt at forgery of identity, and must protect its own validation algorithms.

(3) The security kernel must be simple enough for effective logical certification, and implemented in a small set of simple primitive operations.

In addition, Schell stated a robustness criterion for adjudging the effectiveness of kernel operation: it must be "...so designed that even an antagonist could provide the remainder of the system without compromising the protection provided."

The need for these principles can be seen from the findings of the OS/360 penetration study (4), where lack of a centralized, simple and certifiable kernel was adjudged to be the source of the system's vulnerability.

In Chapter 2 of this paper we propose a conceptual model of the kernel's operation and organization, and in Chapter 3 we particularize the model to the military security problem. Our main objective is to design a logically verifiable kernel subsystem to guarantee operating security.

1.2 Design Methodology

We imagine the design process to move from needs to implementation in a series of levels of abstraction, each level moving closer to a concrete machine realization. The topmost level, level zero, consists of a mathematical model of general protection mechanisms, independent of particular security requirements. The mathematical objects in the level zero model are functions which are expressed in terms of (virtual) primitives unanalyzed at level zero. That is, at level zero, the security kernel is factored into a number of components (modules). Some components remain to be analyzed further in lower levels of abstraction; level zero describes how the components synthesize to achieve the system goal. At the same time, subgoals are established for each of the unanalyzed modules.

At level 1, the process is repeated on each of the modules unanalyzed at level zero. Level 1 codifies the specific requirements of a military security kernel, and identifies still "smaller" unanalyzed primitive components to be analyzed and factored at still lower levels.

The term factorization is appropriate for this process, since at each level the unanalyzed components do indeed compose with other functions in order to realize the goal for that level.

This top-down design approach allows us to make design decisions in an orderly manner -- the choice of factors or modules at each level, and the scheme of synthesis amount to design decisions, and determine constrained subgoals. The approach allows us to separate issues germane to protection from those which are particular to a machine or system.

By far the most important advantage of this approach is that it allows for orderly verification of each level. Verification proceeds in a "bottom up" manner at each level: assuming that the unanalyzed modules behave as hypothesized, then the synthesized function at the level does such-and-such. Having verified the level, the inductive subgoal is now to factor and verify the modules.

At level 2 we envision describing level 1 in terms of a MULTICS-like file directory hierarchy. The notions of directory, segment descriptor and process descriptor are introduced, but "paging" is invisible at this level. Our task at level 2 will be to show that a file directory hierarchy structure realizes the access retrieval function defined at level 1.

Levels zero and one are described in detail in the following chapters. We believe that the utility of the chosen design methodology is illustrated in these chapters.

CHAPTER 2. LEVEL ZERO

2.1 A Kernel Model

2.1.1 ~~The Accession Relation~~ Components of the Model

We employ a protection model based on the work of Lampson (7) and Graham and Denning (6). We have a set of security _objects_ O (files, programs, devices, etc.) a subset S of the set O of _subjects_ (processes) and a set of A of _access attributes_ ('read', 'write', 'control', 'owner', etc.). Access of subjects to objects is controlled by an _accession relation_ R which is a subset of S x O x A. For example, R (s,o,a) or "(s,o,a) in R" is intended to convey that s has attribute a with respect to object o. Lampson regards R as _represented_ in the form of a matrix.

m: S x O -> P(A)

with entries in the power set P(A) of A. We wish to postpone questions of representation until later.

In this model we assume that the accession relation involves a single subject and a single object. That is, we allow in our system a relation like

(1) "s1 can 'read' o1"

but not

(2) "s1 can 'read' o1 only from device o2"

which would involve three objects.

Associated with each type of object is a _monitor_, a program which actually performs the desired accession. Here we wish to illustrate the difference between our visualization of the security system and that of Graham and Denning.

In their model, depicted in FIGURE 1, when a subject s initiates access a to object o, the system supplies the triple (s,o,a) to an appropriate monitor. The monitor interrogates the accession "matrix" to determine whether s has a access to o and, if so, the monitor performs the requested function.

We wish, on the other hand, to propose a picture which isolates the process of access attribute checking, and which separates this process from the monitor functions. This effectively factors out the following processes: the interpretation of a subject request with attendant system mediation, the search for and retrieval of access attributes (which depends upon the way in which the information of R is stored), the checking of retrieved attributes against the accession request, and finally the operation of individual monitors. The situation is depicted in FIGURE 2.

Below we address the questions of design and certification of the Access Evaluator (E), the Attribute Retriever (F), and the Update Monitor (U).

We do not investigate the operation of G, the Access Request Generator. It is the mechanism which guarantees system mediation in all requests for protected objects. As the entryway into the kernel, G must provide a requesting subject with a nonforgeable identification interpreted by the kernel.

Neither can we consider the operation of the monitors. Being concerned with security, we are fundamentally interested in forbidding unauthorized access to any supervisory module. Thus we do not address the possibility of faulty operation of the monitors themselves. The correct operation of the security checking mechanism should guarantee that no program can access the ill-gained fruits of a monitor bug.

2.1.2 <u>Accession</u>

When a program requests an access to a monitor, it requests that a service be performed for it by the system. As such it actually requests an entry to the monitor program.

We imagine the kernel to interpose itself between the requesting program and involved monitor. The kernel must interpret the type of request, identify the objects involved, and perform the requested action or a violation recovery action. It is the function of the Access Evaluator E to retrieve appropriate data, grant or deny the request, and transfer to the appropriate monitor (the violation handler is considered a separate monitor). In

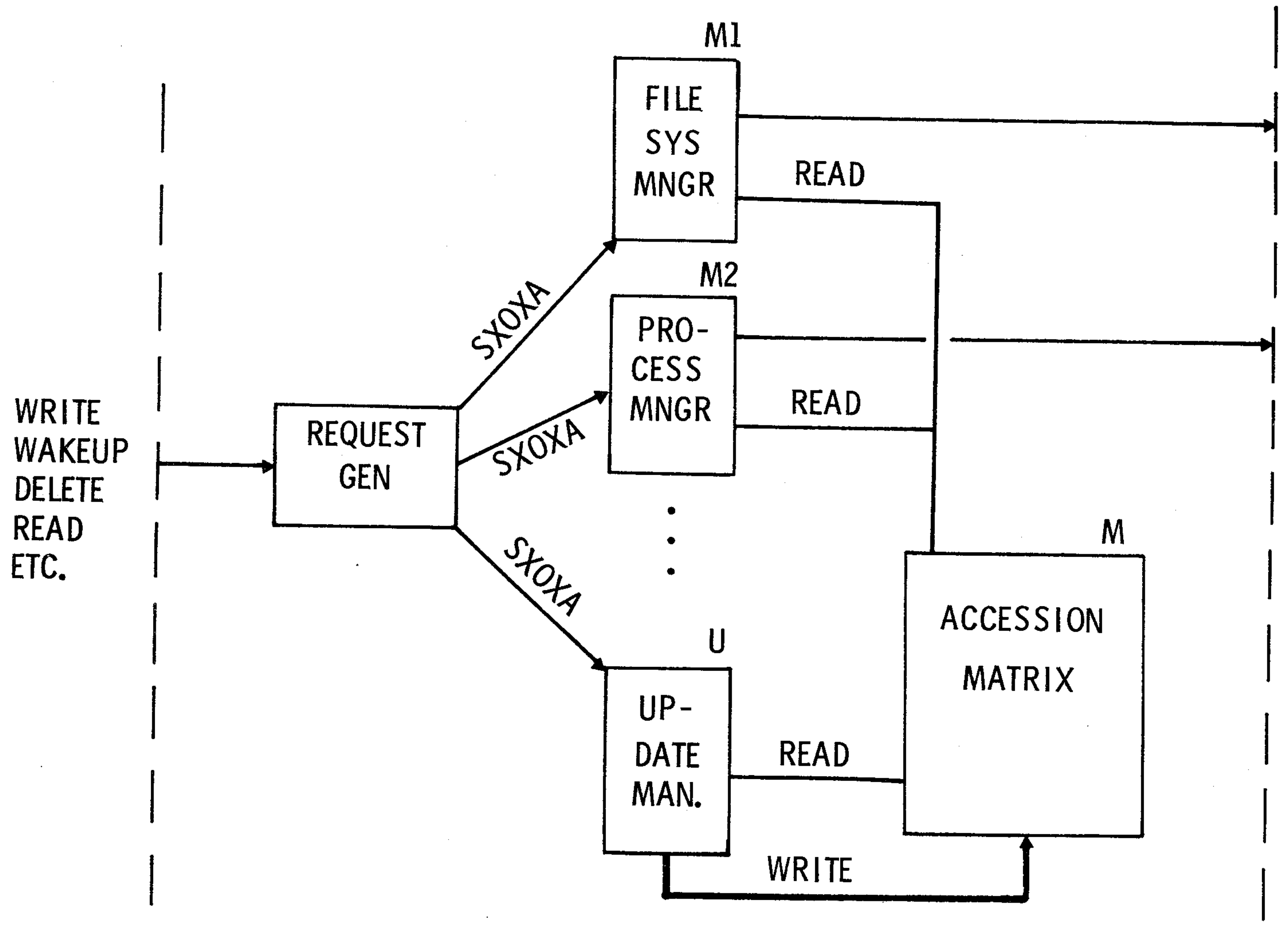

FIG. 1. GRAHAM-DENNING MODEL

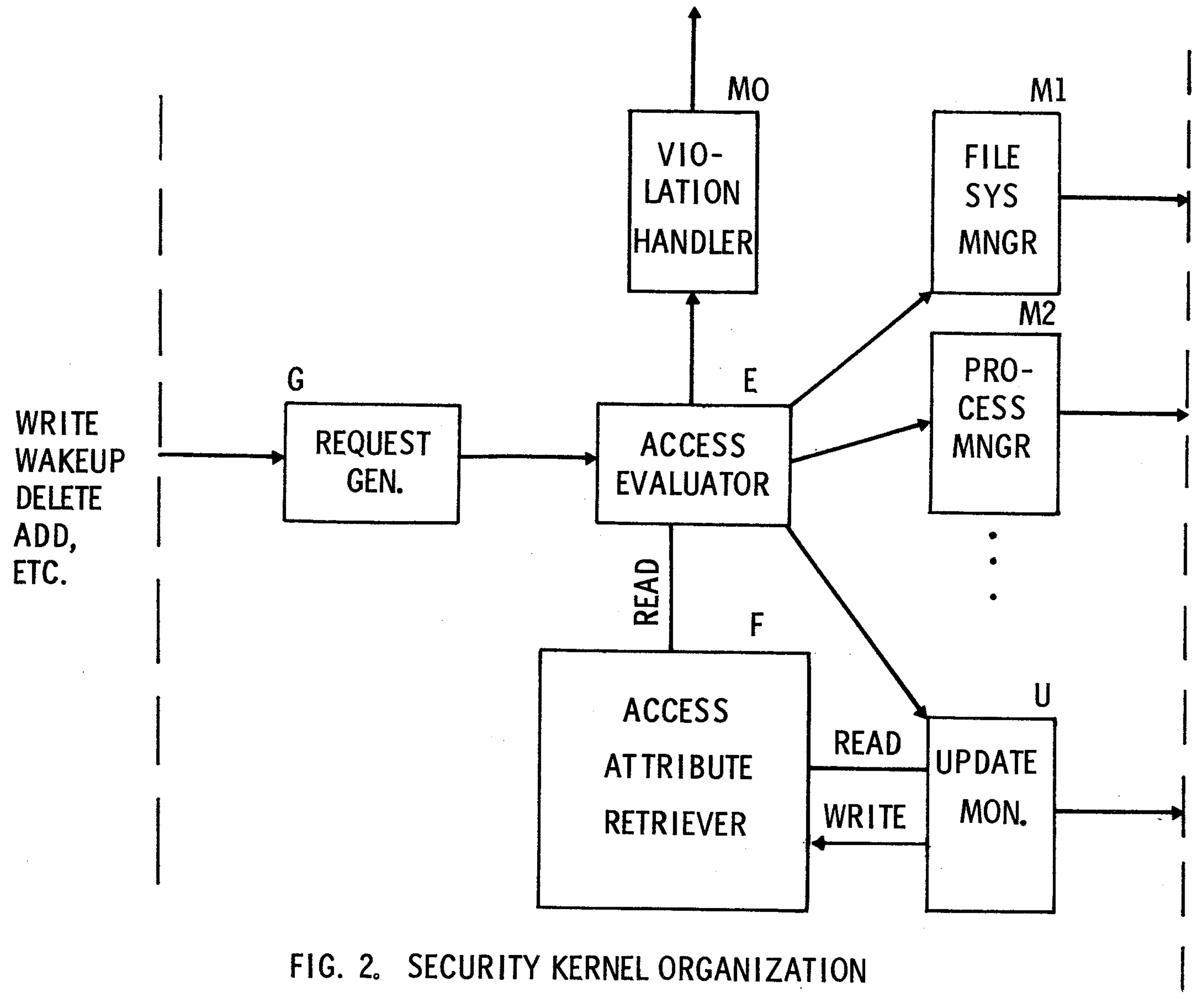

FIG. 2. SECURITY KERNEL ORGANIZATION

our description of the function performed by E, we say that _function_ e returns a _function_ mx, where mx is the function performed by monitor Mx. We use upper case to denote programs (system modules); corresponding lower case to denote the actions (functions) which they perform.

Let e be the function realized by program E, and let f be the function accomplished by the program F which fetches the accession data. Then

$$e: \quad SxOxA \rightarrow \{m(0), m(1), .., m(x)\}$$

$$f: \quad SxOxA \rightarrow B \ (1)$$

tells us some information about e and f -- their _types_. It does not explain the details of their action, but shows at least the nature of their inputs and outputs.

Given the required accession relation R, F operates _correctly_ if we can guarantee that

(1) $\forall s. \forall o. \forall a. \quad f(s,o,a) = 1$
$\quad\quad$ iff $R\ (s,o,a)$.

This just states that F has correctly stored and correctly retrieves the accession data.

Suppose h(o,a) retrieves the index of the proper monitor function m(h(o,a)) among m(1),...,m(x). For example, if o is a data file and a is 'read', then h(o,a) is the index of the file system manager.

Assuming f,h are verified to operate correctly, then e can be correctly realized by

$$e(s,o,a) = \underline{\text{if}} \quad f(s,o,a) = 1 \ \underline{\text{then}} \\ m(h(o,a)) \ \underline{\text{else}} \ m(0).$$

Notice that m is a mapping which takes the name of a monitor program and returns the _mapping_ realized by this program. In an implementation this might mean: fault to an appropriate location in a protected program.

(1) B = {1,0} or {_true_,_false_} is the set of _bits_.

Summarizing, we have the maps

f: $S \times O \times A \rightarrow B$

h: $O \times A \rightarrow \{1,2,\ldots,x\}$

m: $\{1,2,\ldots,x\} \rightarrow \{m(1),\ldots,m(x)\}$

m(i): unspecified functions

The arguments and values of the m(i) (their _types_) depend upon their respective duties, and clearly involve data external to the kernel. For example, the memory addressing hardware monitor will need to know where in the requesting program to return the contents of an address. By leaving these details unspecified, the most we can say is that e returns one of a finite set of explicit functions. Further analyses may now enumerate, factor and describe the implementation of the m(i). What we have done is to get them out of the access-checking game, concentrating on their "natural" roles.

Returning to the question of certification, what if (1) is violated, i.e., f does not adequately reflect the desired accession relation R? Popek (9) has suggested (the _inclusion technique_) that we add to the antecedent of e

_if_ f(s,o,e) = 1 _then_ ...

all of the extra checks demanded by R, after writing a suitable routine d to store and retrieve these checks. We will then have another program

e'(s,o,e) = _if_ f(s,o,e) =1 _and_ d(s,o,e) =1

then ...

which will now operate correctly. But this amounts to constructing a retrieval function f' satisfying

f'(s,o,e) = f(s,o,e) _and_ d(s,o,e).

What is evidently needed is a correct retrieval function f satisfying (1). In a practical system, it is the structure of f which is of utmost importance anyway.

In a later section we analyze the function f with particular emphasis on a military security model, and introduce the notions of locks and keys in the operation of f.

2.1.3 Updation

As the system evolves over time, the security state, represented by the accession relation R, is modified by the attentions of the update monitor U. Available for the use of subjects are various security state updation commands (delete, grant, destroy, etc.) which request changes to attributes, destruction of subjects and objects, etc.

A destroy subject s2 command by subject s1 is, for example, interpreted by G as a request to write to the (protected) accession data F. The information (s1, F, 'write') is passed to the Access Evaluator E which determines whether s1 is allowed to change any items at all in F. If control is passed to U, U must now do further careful checking to:

(a) retrieve the name of the particular subject s2 which is to be destroyed;

(b) determine whether s1 is allowed to destroy s2;

(c) perform the desired action, or else refuse, with attendant action.

Operation (a) is performed with no difficulty, but operation (b) requires some explanation. The internal logic of U determines the type of access attribute s1 needs vis-a-vis s2 in order to destroy s2, say 'owner'. U then interrogates F with the request (s1,s2,'owner'), and if this triple is part of the current security state, U then updates F in the required fashion (deleting s2 from the data base) and passes control to further non-kernel systems for housekeeping duties.

The monitor U, being inside the kernel, is "trusted" by E, and there is no need for U to go through G or even E in order to access F. Hence U may successfully "disguise" itself as s1 for purposes of reading s1's privileges.

Other types of updation can be handled in a similar fashion.

Notice that in this model the detailed study of F-updation privileges is done by U, which must take into consideration a larger context of information than E; but U has F at its disposal.

The correctness of an implementation of U depends upon a full description of the circumstances under which the system is to honor a request to alter the security state. These circumstances are imposed on the design from without, in the form of a set of <u>updation constraints</u>. It must be demonstrated that any change which U makes results in an acceptable security state within the updation constraints.

Updation constraints are described in a logical language, and codify just those rules which the designer wishes to place upon U in its making of updation decisions.

Associated with each updation command is a <u>predicate</u> true if and only if the command can legally be executed by the requesting process.

<u>Example</u>: <u>delete</u> ('read',s2,o) is a command uttered by s and asking that attribute 'read' be withdrawn from s2 vis-a-vis o. There is an associated predicate

$$DEL \subseteq S\times A\times S\times O.$$

DEL(s1,'read',s2,o) is true if and only if s1 is allowed to delete s2's 'read' privilege to o.

An example of an updation constraint is:

$$\forall s1.\forall s2\ [R(s1,s2,\text{'control'}) \dashrightarrow \forall o.\forall a\ DEL\ (s1,a,s2,o)]$$

which says in words "For every s1 and s2, if s1 has 'control' access to s2, then for all objects o and attributes a, s1 can delete s2's a-access to o." Or better: "If s1 has 'control' of s2, then s1 can delete any of s2's privileges to any object."

Other examples follow which may readily be translated by the reader:

$$\forall s1.\forall o.[R(s1,o,\text{'owner'}) \rightarrow \forall s2.\forall a.GRANT(s1,a,s2,o)]$$

$$\forall s.\forall o[R(s,o,\text{'owner'}) \rightarrow DESTROY\ (s,o)]$$

Within U are a series of programs, called _updators_, W1,...,Wz. These effect the actions requested in the commands by users. The actions they perform are denoted w1,..,wz. U also has as a factor a program V, the _Constraint Checker_, which matches an updation request against the updation constraints, suitably internalized. FIGURE 3 illustrates the situation, and FIGURE 3a shows the parallel nature of E and U.

V employs the retrieval program F to make its decisions. Its internal logic should be designed from the (fixed) updation constraints as outlined above. Obviously its operation cannot be illustrated without a predefined set of constraints to work from. However we can give an

Example Suppose we have the updation constraint

$$\forall s.\forall o.R(s,o,\text{'owner'}) \rightarrow DESTROY(s,o)$$

and let W2 be the "destroyer" program. Then the description of V will include in part the line

```
... if f(s,o,'owner') = 1
    then w2 ...
```

As in the case of e, v outputs one of a set of functions w1,...,wz.

Now providing that V operates according to the given constraints and provided that the individual updators perform their assigned tasks correctly, U will operate correctly, and the system will never enter a state which compromises security. Why? A correct, satisfactory, or secure system is _defined_ by the set of updation constraints. U merely enforces them.

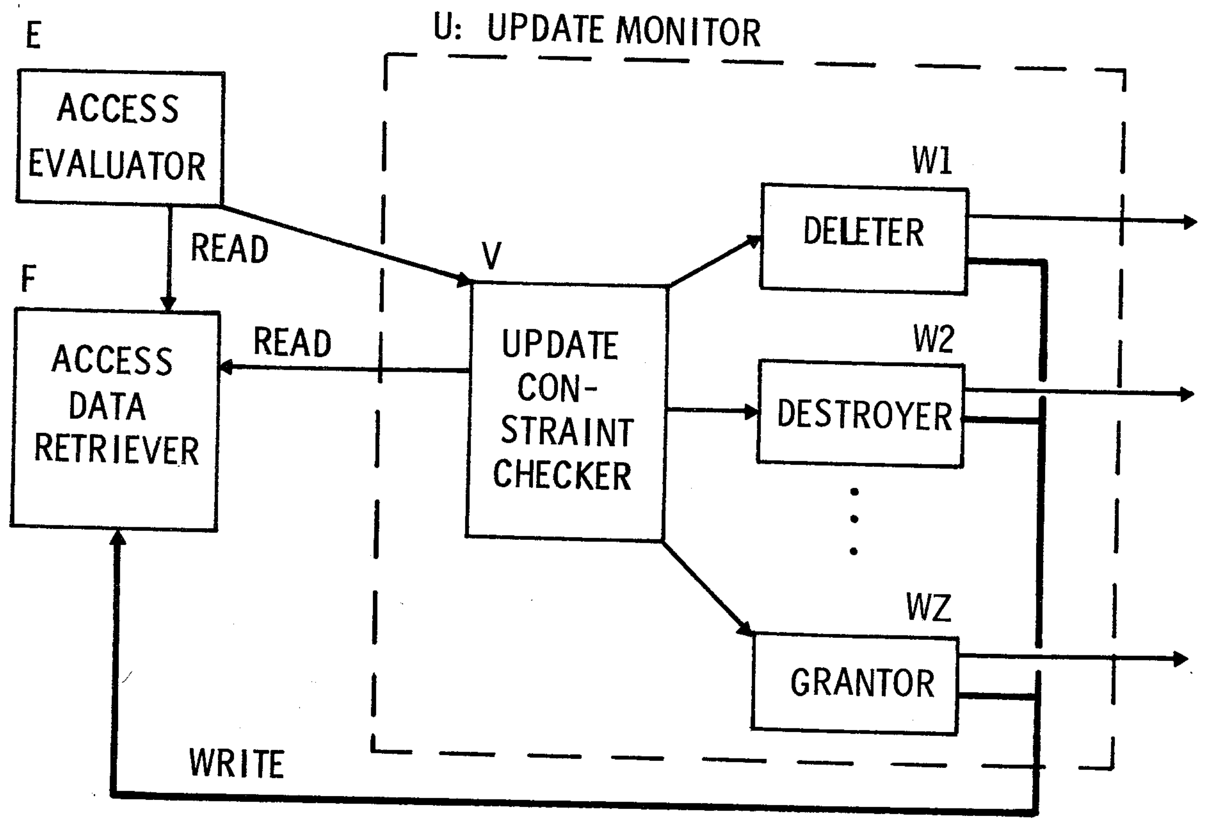

FIG 3. UPDATE MONITOR

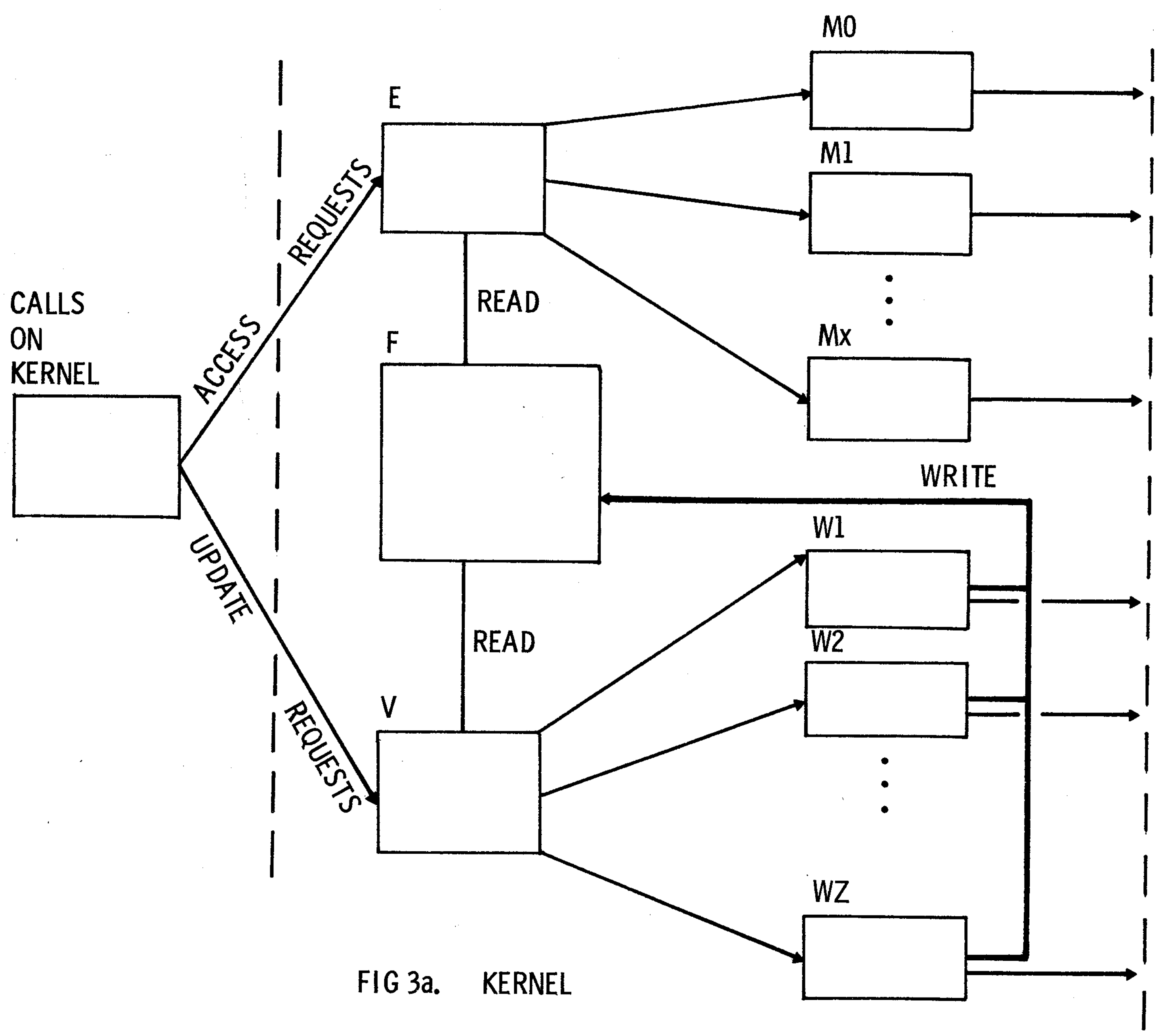

FIG 3a. KERNEL

All the above assumes that the constraints form a consistent set. For an example of an inconsistent constraint set, imagine both

"s1 may never a-access o3"

and

$\forall s[\mathrm{GRANT}(s0,s,a,o3)]$.

Then it is clear that s0 could, quite innocently, grant s1 access a to o3.

No kernel or system will ever be able to enforce an inconsistency.

Clearly a design prerequisite is a complete description of commands and their logical inter-relationships, expressed in the updation constraints. This "requirements" list must first be checked for consistency. Provided it is so, V may be encoded to check that each constraint is satisfied for each updation request. This provides another example of Popek's (9) Inclusion Technique.

2.2 Modeling Access Data Retrieval

In this section we focus upon the operation of the retrieval program F. We give some attention to the possibilities of factoring this program into (perhaps) more simply verifiable components.

The function of F is to represent and retrieve the information contained in the accession relation R - the security state of the system. Since the numbers of subjects S, objects O, and access attributes A are all finite, R is in principle "just a big table" and F "just a big table look-up". This might satisfy an automata theorist but not a systems designer.

(1) First, there is an enormous amount of information contained in R -- the triples (s,o,a) not in R are as important as those in R.

(2) Second, R is sparse as a table, or even as a matrix

$r: S \times O \to P(A)$,

suggesting that in practical cases a great deal of structural constraint obtains among the entries.

(3) Third, the amount of information is so large that present-day systems employ both dynamic and static storage techniques in its retrieval. For example, in the MULTICS segmented memory, with its dynamic linkage facility, part of the protection information is stored in the environment of a process; the rest is distributed throughout the storage system and available for later dynamic recall. In future systems there may well be a requirement to segment this information base.

(4) Fourth, people group and use information according to behavior patterns and in established structures. For example, very few systems development programmers call a linear regressions package, and many data files group naturally together with the associated project which developed them.

For all these reasons, we believe in careful structuring of the program F with a view toward certifying its operation. Below we make a start toward this analysis.

2.2.1 A Model for Data Configuration

2.2.1.1 Mathematical Language

In this model we do not discuss issues of implementation, but do wish to develop a theory for the structuring of data used in the security kernel.

From our point of view, set theory is not an adequate tool for the expression of notions in computing. The most primitive semantic notion for programming is that of function or mapping. A set, as a primitive, orderless collection can never be realized on a computer, whereas a function, the characteristic function of the set, can be so implemented. That is, set S cannot be "in" the machine, but a map c: S->B can be realized as a bit string if the size of S is small; as a linked list if large, etc.

If S,T are finite sets, (S->T) represents the set of all possible maps from S to T. The statement

$$c: S \to B$$

means that c is in (S->B), a set. Whenever we write f: (A->B)->C or say "f in ((A->B)->C)", we are expressing the type of f as a mathematical object. This gives only a limited amount of information about f -- its domain and range -- but is frequently useful.

Another concept used all the time is that of cross-product of two sets SxT. Since this is just a set and not representable on a machine, we think of it as a function

$$p: S \times T \to B$$

defined to give 1 for all pairs in SxT.

On a machine we cannot really make sense of an ordered pair. Pairs must be stored, and in some order. We adopt the fact that

$$(S \times T \to U) = (S \to (T \to U)).$$

That is, by convention an S,T matrix of U-values is stored as an S-list of T-lists of U's.

2.2.1.2 Examples

We give here some examples of the way in which F might be arranged as a program.

Ex. a. A security system using Access Control Lists (ACLs).

The accession relation is stored by object, then subject, then attribute. The accession function is

$$f: O \to (S \to (A \to B))$$

Given an o in O,

$$f(o): S \to (A \to B)$$

_is_ an ACL -- the ACL of o -- and is itself a function. Given a subject s, f(o) finds an access attribute list (_function_)

f(o)(s): A->B

associated with o and s. This list might be represented as a bit string. The point is that f(o)(s), given an a, returns a bit. _How_ it does this is implementation. A pictorial diagram of the above might be given for MULTICS in FIGURE 4.

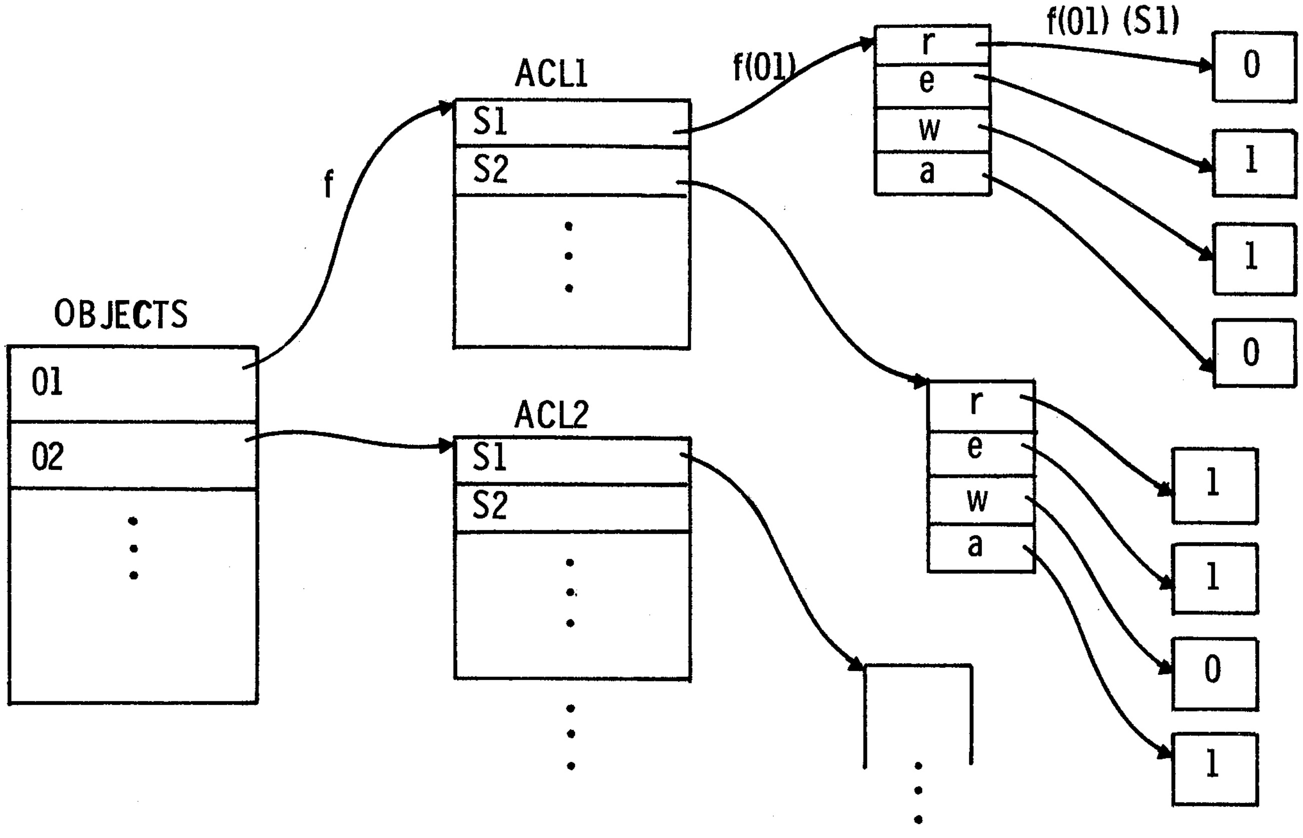

FIG. 4

The bundles of linking arrows represent the _functions_ f, f(o), f(o)(s), and may not be simple pointers, but some complex hashing scheme.

Notice that in MULTICS, the object collection itself has a further structure, the _file_ _directory_ _hierarchy_, not shown here.

_Ex. b._ A System with Capability Lists (C-lists)

R is stored by subject, then object, then attribute:

f: S->(O->(A->B))

The C-list for subject s1 _is_ a map

f(s1): O->(A->B)

which returns, for an object o2 a map

f(s1)(o2): A->B

the attribute list.

An oversimplified example is given from MULTICS in FIGURE 5.

Multics actually consists of a complex of both techniques. Initially the system checks the descriptor segment for an object. If it is not present, a missing segment fault occurs, the file directory hierarchy is searched for the object and the descriptor segment is updated with the object identification and access information.

2.2.2 Generalized Locks and Keys

2.2.2.1 Definitions

Frequently the sparse structure of R or any of its representations discussed above can be exploited to factor the retrieval problem. Indeed, both natural groupings of subjects (think of projects) and natural groupings of objects (think of master files) may exist. The idea of key and lock exploits this observation: why treat each subject or object as a separate security entity, when coarser groupings may be more efficient?

Let K be a finite set of keys, L a finite set of locks. The only thing we require is that these sets consist of distinguishable objects (e.g., bit positions in a word). A _key assignment_ is a map

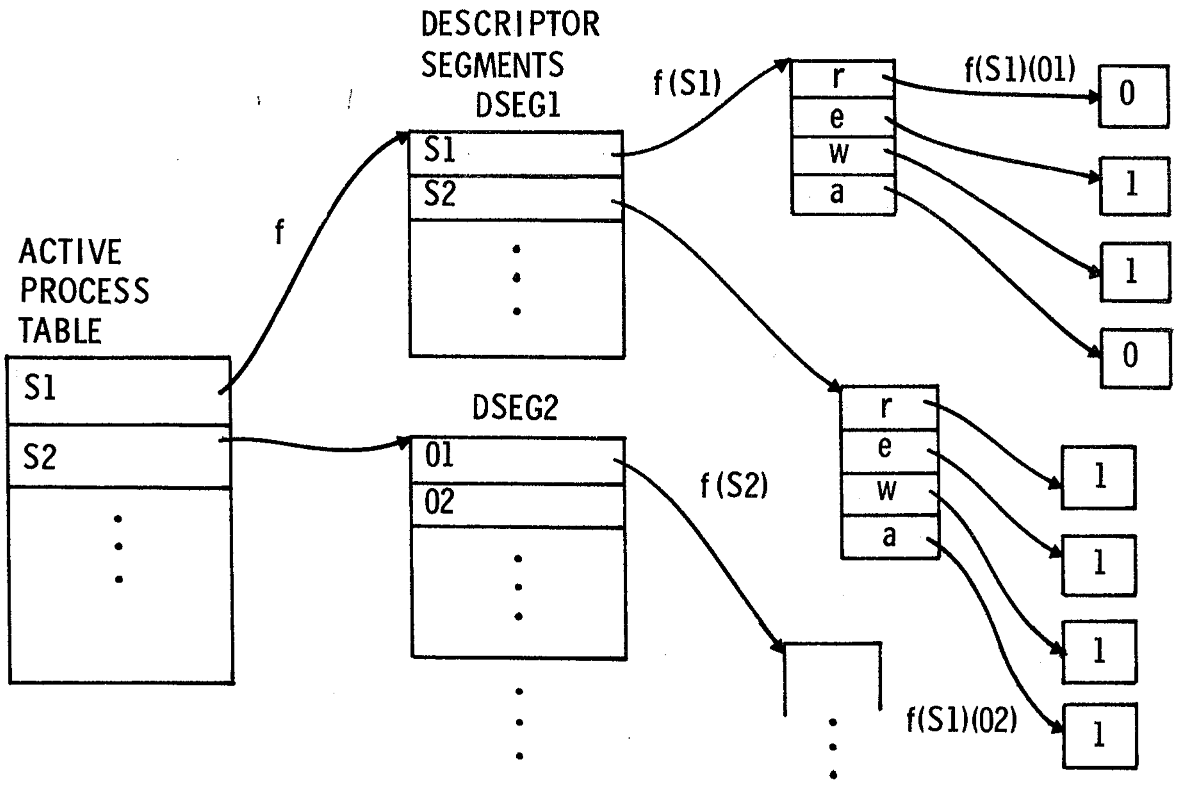

FIG. 5

k: S -> (K->B)

and a lock assignment a map

l: O -> (L->B).

A subject may thus be issued several keys; and an object may have several independent locks.

We also have an unlocking relation which tells which keys are adequate to which locks

t: KxL -> (A->B).

This is not a one-to-one relation, nor even a function; for a passkey may open many different locks, and a lock may be opened by a hierarchy of passkeys of varying power.

Subject s has access a to object o if and only if there are k1,l2 such that

k(s) (k1) =1,
l(o) (l2) =1
and
t (k1,l2) (a) =1.

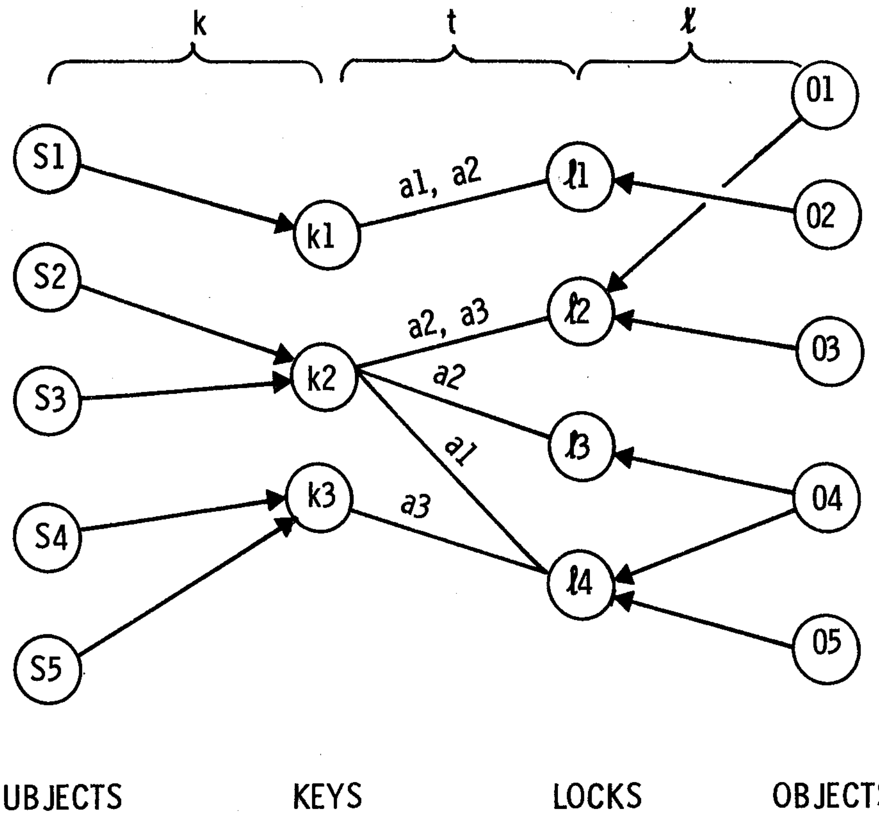

FIG 6.

We may depict an accession relation as in FIGURE 6.

Notice that any accession relation represented using intermediate locks and keys can of course be realized by an accession matrix

$$m: S \times O \to (A \to B)$$

merely by defining $m(s)(o)(a)=1$ if and only if there are k1,12 such that $k(s)(k1)=1$, $1(o)(12)=1$ and $t(k1,12)(a)=1$. But this misses the point. Locks and keys, which look like a fatuous complication in the abstract, are introduced in practice for natural reasons leading to greater efficiency. In an application it may prove more efficient to calculate k, 1, and t than to look up entries in a tree structure such as those of 2.2.1.2.

2.2.2.2 Example: A Military Security Model

In an application, the notions of lock and key may be used to store one component of the accession information, while other techniques are used for the remainder. Possibly complex overlays of various storage representations may be used if efficiencies result. The problem of a military security data base is a good example.

Three factors govern the control of access to protected information.

(a) clearance/classification. A document, file or program (information) is said to be classified U,C,S or TS. A user or subject is said cleared for U,C,S,TS. Below we represent these security levels by integers 0,1,2,3.

(b) compartmentalization. As a refinement of (a), information and users are further assigned one or more compartments, reflecting the kinds of classified information to which they belong or have access. The military employs 16 compartments $P=\{1,..,16\}$, e.g., cryptographic, AEC, etc.

(c) need to know. The finest resolution of the security question occurs at this level. For each subject s and object o, the military requires that some authority grant s an "a-need to know" for o before s can a-access o. Examples might be "need to read", "need to execute", etc. From our point of view, the various "needs to know" are an application of the notion of access attribute for a subject/object pair. The information is stored in the underlying accession relation for the system.

Clearance/classification may be modeled by the following lock/key arrangement (the key/lock functions are denoted by the same symbol in this example).

$$c: S \to C$$
$$c: O \to C$$
$$t: C \times C \to B$$

where $C = \{0,1,2,3\}$ and

$t(i,j)=1$ if and only if $i \geq j$.

Compartmentalization is represented by

$$p: S \to (P \to B)$$
$$p: O \to (P \to B)$$
$$z: (P \to B) \times (P \to B) \to B$$

where $z(p(s),p(o))=1$ if and only if $p(s)$ and $p(o) = p(s)$. Here and represents the bit mask of lists.

Finally, as noted above, need-to-know must be handled by explicit retrieval, structured however is convenient. We depict the situation in FIGURE 7.

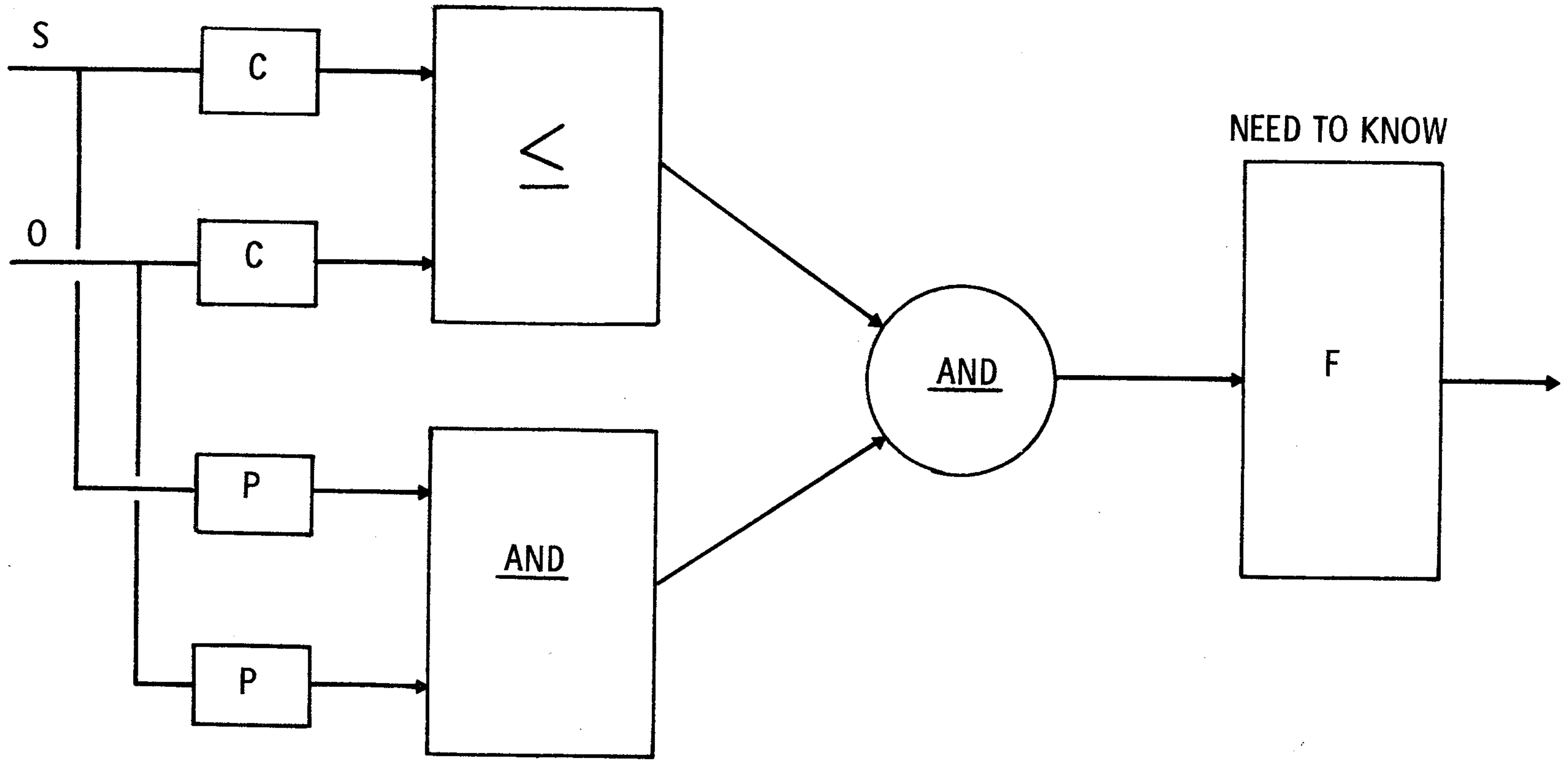

FIG 7. MILITARY SECURITY DATA BASE

CHAPTER 3 A MILITARY SECURITY MODEL

The purpose of this chapter is to propose the requirements of a military time-sharing system operating in multilevel security mode. DOD 5200.28-M defines multilevel security mode as:

"A mode of operation under an operating system ... which provides a capability permitting various levels and categories or compartments of material to be concurrently stored and processed in an ADP system. In a remotely accessed resource-sharing system, the material can be selectively accessed and manipulated from variously controlled terminals by personnel having different security clearances and access approvals..."

The model will be independent of implementation in the sense that it will be possible to interpret the rules of the kernel as being enforced by a human security officer handling documents, not necessarily by a computing system. The model will be formulated using existing military security requirements for document control (AFM 205-1), as well as requirements which have been established for existing military computer systems (WWMCCS GCOS, DOD 5200.28-M). Below, in referring to the <u>manual system</u>, we shall mean present military procedures for physically handling classified documents, as specified in AFM 205-1.

3.1 General Considerations.

3.1.1 <u>Three Dimensions of Security</u>

In the military three factors control access to protected information, as discussed in section 2.

3.1.1.1 <u>Clearance/Classification</u>

Possible clearances are $\{C = 0,1,2,3\}$. With each object is associated a clearance or classification via the map.

$$c:O \rightarrow C$$

3.1.1.2 Compartmentalization

Compartments are $P=\{1,2,\ldots,16\}$. Each object is assigned to a list of compartments by the map.

$p:O\to(P\to B)$

Thus if o is in O, $p(o): P\to B$ is o's compartment list.

3.1.1.3 Needs-to-Know

We regard this as equivalent to the notion of access attribute. Given an attribute set A (discussed below) the function $f:O\to(S\to(A\to B))$ assigns to each object o:O a list of needs-to-know $f(o):S\to(A\to B)$ classified by subject. Notice that we are proposing an Access Control List structure for f (Cf. section 2.2.1.2).

Evidently clearance and compartment information could be stored implicitly in the access retrieval function f (Cf section 3.2.3). However, for purposes of access checking and updating this would neither be efficient nor would it model the existing military manual system. As a consequence we factor the accession data as indicated.

3.1.2 System vs. User Responsibility

In designing requirements for a military multilevel security system, we must decide at the outset the role of the kernel in transactions involving secure data.

a) The Responsible Kernel

The kernel itself is responsible for the control, classification, declassification and manipulation of information within the system. It employs automatic rules to assign classifications to newly created files, maintains a history of each user's security environment and watches each user to maintain operating consistency. This approach is illustrated in Weismann's paper (11). The kernel from this point of view becomes a super bureaucrat.

b) <u>The Responsible User</u>

The assumption is that an authorized user of classified information has full responsibility for its control while operating with it. Thus destruction of copies, reclassification of altered files, etc., become duties of the user which must be performed before he logs off. The kernel, after granting initial access, makes no attempt to monitor the use to which data is put.

Whichever role the kernel is designed to play, the system of rules which the kernel enforces must be simple enough and so clearly stated that each user understands the full implications of each security state updation command, and his responsibilities in employing it.

The assumption of <u>user responsibility</u> is the one which agrees most readily with the present manual system, and will underlie the design discussed here.

As a consequence of the "responsible user" assumption, certain possible "security compromises" of concern to Lapadula and Bell (8) are neither detected nor prevented by our proposed system. To use their example, suppose s1 is cleared for TS, s2 for S and let file o3 be classified S. Suppose s1 writes some top secret information in o3, but fails to explicitly upgrade o3's classification. The kernel cannot detect the "violation". At some future time, s2 could be granted 'read' access to o3, and s2 would be reading "forbidden" information.

Our feeling is that any attempt to make the kernel responsible for detection and prevention of such occurrences would either (i) involve the kernel in deciding complex questions of sensitive data aggregation, or would (ii) require the adoption of an arbitrary "high-watermark" rule (e.g., s1 operating under a TS clearance can only write TS files). The latter approach is adopted by Weissman (11), who does not allow for the possibility of declassifying files.

Here we only require the kernel to enforce existing manual security regulations which place the onus of responsibility upon the user of a document to make necessary changes to its classification or compartments. Since we demand that the kernel allow reclassification on some authority, compromises of the kind illustrated above will always be possible on some level. We have chosen to trust completely every authorized user. The kernel is non-suspicious -- if a subject is granted access rights by the kernel, the subject has the full implications and responsibilities attendant on those rights.

Another, more technical, way of phrasing this is that the kernel uses only subject/object ID's, classifications, compartment lists, and access attributes in reaching its decision. The kernel does not interpret or deduce any implications from an authorized access or update request.

For example, if s1 accesses an object o2 for which it has inadequate clearance, a security violation occurs. But if s1 obtains upward reclassification from an "incompetent" but authorized subject s2, and then accesses o2, no violation, from the system standpoint, has occurred.

3.1.3 Separation of Accession and Updation

As discussed in the previous chapter, the processes of granting "normal" access, and the granting of updates must be kept distinct, since the latter action is more complex. It follows that the data used and modified by the updation procedures, the accession relation R, should be kept distinct from ordinary protected data files. For one thing, it will have to be maintained in a rigid format interpretable by the kernel. For another thing, it is part of the kernel itself, since its compromise would compromise the entire system. Lastly, it may be stored in a radically different manner - perhaps in special hardware.

In our model this data is stored in the access data retrieval program (F). We see no reason to treat it as an object (compare Popek's (9) security objects), since it deserves such special status.

3.1.4 'Control' and 'Owner' Access Attributes

The notions of 'control' and 'owner' access attributes occur in Lampson (7) and Graham and Denning (6). One subject s1 'controls' another, s2, if s1 can read from and write in s2's row of the matrix m:S ->

O->(A->B), i.e. if s1 can read and modify s2's capabilities. If, in addition, s1 may destroy s2 or grant to other subjects any access to s2, then s1 is said to have 'owner' access to s2. Thus s1 'owns' s2 when s1 may read from and write in s2's _column_ of the access matrix. Issues immediately arise concerning multiple 'owners' and the transferability of 'control', which are surveyed by Graham (5).

We shall not introduce these attributes. The relation of s1 'owning' s2 can be replaced by granting s1 all possible attributes for s2. Obviously then, multiple 'owners' are possible.

If s1 can 'control' s2, this implies that s1 can obtain and modify all s2's _capabilities_ - the list of all objects _to_ _which_ s2 has access. In our model, access attributes will be stored in ACL form (Section 2.1.2, example a). There is no way for s1 to conveniently learn s2's privileges, short of listing all objects and requesting the ACL of each. (This is exactly the situation in MULTICS.) We see no apparent reason for introducing the 'control' facility.

Furthermore, in the military manual system, possession of document implies "control" of it and responsibility for it. A possessing subject can give it away, garble it, etc.

We choose to introduce the simple attribute 'update'. Subject s1 with 'update' attribute for o2 (subject or not), may modify the security data concerning o2 (access attributes _to_ o2, clearance, compartments). There will be no facility for one subject s1 to affect a second subject's attributes vis-a-vis an object o2, unless s1 has 'update' permission for o2. When s1 has 'update' permission for s2, s1 can only limit accesses by other subjects _to_ s2.

Update permission may be passed to other subjects like any other attribute.

3.2 Elements of the Model

3.2.1 <u>Access Attributes</u> (A)

The set A consists of five attributes

$A=\{r,e,w,u,l\}$

with the following meanings:

<u>Attribute a</u>	<u>if f(o)(s)(a)=1 then</u>
r	s can <u>read</u> the contents of object o, implying that s can copy o.
e	s can <u>execute</u> the (executable) object o. s must know the calling sequence for o, since s cannot read o.
w	s can <u>write</u> to o, altering it, adding to it, even zeroing it out.
u	s can <u>update</u> (write on) the descriptor (see section 3.2.4 below) of o, adding to it or deleting from it.
l	s can <u>look</u> at the contents of the descriptor (see section 3.2.4 below) of o, without affecting its contents.

3.2.2 <u>Modes</u> (K)

The mode (1) of an object is an indicator of the kind of object it is -- terminal, process, data file, directory, etc. Depending on the characteristics of the computer system, there may be different modes, each usually associated with a special subsystem or monitor for handling objects of the same mode. We choose a mode set

(1) Called by Burke (2) a <u>type</u>. We have used type in a more technical sense, so we employ Popek's (9) term mode.

$K=\{t,p,f,d\}$

and a function k:O->K assigning to each object an unique mode, with the following meanings:

Mode Kl	if K(o)=Kl then o is
t	a terminal
p	a process, i.e., a subject
f	a file, i.e., a protected block of data not interpretable by the system.
d	a directory, a specially formatted file which may be interpretable by the system.

Other modes may be introduced depending upon the particular system.

3.2.3 Access Data Retrieval (F)

In the military security model, the data used by the kernel to determine privileges is stored in a factored accession matrix, as in section 2.2.2.2. We represent it by the three functions

$f:O\to(S\to(A\to B))$

$c:O\to C$

$p:O\to(P\to B)$

where $C=\{0,1,2,3\}$

$P=\{1,\ldots,16\}$

$A=\{r,e,w,u,1\}$

There are three different relations, all devoted by $\leq$, which will be useful below:

(i) $\leq$: $C \times C\to B$

denotes the usual inequality on integers.

(ii) $\leq$:(P->B) x (P->B) -> B

denotes the subset relation on the _compartment lists_; object r is a member of (P->B). (1)

(iii) $\leq$: (A->B) x (A->B) -> B

denotes the subset relation on _access lists_; object a is a member of (A->B).

A convenient abuse of notation will allow us to identify sets in P(A) with functions in (A->B). For example, $\{u\}$, which usually denotes the singleton _set_ $\{u\}$ in P(A), will mean for us the _function_ $\{u\}$:A->B given by

$$\{u\}(x)=1 \text{ if } x=u \\ \quad\quad 0 \text{ if } x\neq u$$

Either point of view is seen to be equivalent, but we believe that the "list" notation (A->B) is more suggestive.

3.2.4 Descriptors

A useful auxiliary notion is that of _descriptor_ of an object, as used by Popek (9). For each o in O, d(o), the _descriptor of O_, is a quadruple of functions

d(o)=(c(o), p(o), f(o), k(o))

or, equivalently

d(o)(1)=c(o)

d(o)(2)=p(o)

d(o)(3)=f(o)

d(o)(4)=k(o)

(1) The notations P(A), 2 exp A, and (A->B) may all be considered equivalent. We use (A->B) because it reminds us we are dealing with _functions_.

Thus d has type

$$d:O\to C \times (P\to B) \times (S\to(A\to B)) \times K$$

This is one way to model the storage of access data. A descriptor is a sort of generalized Access Control List (ACL), and is particularly appropriate when a MULTICS-like file directory hierarchy is contemplated. Descriptors are then naturally stored as elements of directory segments.

While at this stage nothing forces us to introduce the notion of descriptor, it will be convenient.

3.2.5 The Access Evaluator (E)

Normal accession requests, not involving updation, pass through E, whose function is easily described. In our informal programming language, we shall be sure to declare the types of all functions mentioned in the program. Let $M=\{m(0), m(1),\ldots m(x)\}$ be the set of monitors, m(0) the violation handler, m(1)=V the access checker. Let $h:O \times A\to\{1,2,\ldots x\}$ be such that h(o,u)=1 for all o in O.

```
e(s1,o1,b)
        e: S x O x A->M
       s1: S
       c1: O
        b: A
        f: O->(S->(A->B))
        c: O->C
        p: O->(P->B)
        h: O x A->{1,2,...}
        m: {0,1,2,...}->M
        ≥:C x C->B
        ≥:(A->B) x (A->B) ->B
        ≥:(P->B) x (P->B) ->B
        if c(s1)≥c(o1) and p(s1)≥p(o1)
           and f(o1)(s1)≥{b}
        then m(h(o1,b))
        else m(0)
end e
```

3.2.6 Updation Commands

A user program desiring to effect changes to the descriptors requests the kernel to perform the service for him by issuing an updation command. The updation program verifies the user's authority to make the change, and performs the service for him using its updators.

The commands and their intents are:

Command	Intent
write (o,s,a)	sets the access list f(o)(s) to a:A->B, destroying the previous list.
read (o,w)	writes clearance, compartment, access list and mode of o in w.
clear (o,n)	sets the clearance of o to n, destroying the previous value.
compt (o,r)	sets the compartment list of o to the list r:P->B, destroying the old list.
create (o,z)	creates an unique ID for o and associated descriptor with C(o)=0, p(o)=∅ full access privileges for creating subject and K(o)=z.
destroy (o)	nullifies the descriptor of o, erases the ID and the object contents.

3.2.7 Updators (Wi)

These are the kernel programs which actually perform operations on the descriptors, and which call any further system monitors needed for allocation, garbage collection, etc. There will be an updator corresponding to each command:

wr, rd, cl, cp, cr, and ds

The constraint checker V calls the updators, as illustrated in the next section.

3.2.8 The Update Monitor (U)

In the programs below we shall not again declare c,p,f,$\leq$,m(0). Two functions mentioned below make(o) and break(o) are left undefined. They are responsible for housekeeping duties associated with creation and destruction of objects

```
V(s1,request,o1,s2,a1,w,n,z,r)
       s1:S
       s2:S
       o1:O
  request:{write,read,clear,compt,create,destroy}
       a1:A->B
        w:C x (P->B) x (S->(A->B)) x K
        n:C
        r:P->B
        z:K
  if request='write'then
     begin
         if c(s1)≥c(o1) and p(s1)≥p(o1) and
         f(o1)(s1)≥{u} and c(s2)≥c(o1) and
         p(s2)≥p(o1) and not (o1=s2 and a1≥{u})
            then wr(s2,o1,a1)
         else m(0)
     end
  else if request = 'read' then
     begin
       if c(s1)≥c(o1) and p(s1)≥p(o1)
           and f(o1)(s1)≥{l}
            then  rd(o1,w)
            else  m(0)
     end
  else if request = 'clear' then
     begin
       if c(s1)≥n and c(s1)≥c(o1)
         and p(s1)≥p(o1)
         and f(o1)(s1)≥{u}
       then  cl (o1,n)
       else  m(0)
     end
  else if request = 'compt' then
     begin
       if p(s1)≥r and c(s1)≥c(o1) and f(o1)(s1)≥{u}
       then  cp (o1,r)
       else  m(0)
     end
  else if request = 'create' then cr(o1,s1,z)

  else if request = 'destroy' then
       begin
          if c(s1)≥c(o1) and p(s1)≥p(o1)
          and f(o1)(s1)≥{u}
          then  ds(o1)
       end
  else  m(0)
end V

wr (s2,o1,a1)
       s2:S
       o1:0
       a1:A->B
       f(o1)(s2)<-a1
end wr

rd (o1,w)
       o1:0
        w: C x (P->B) x (S->(A->B)) x K
        w <- (c(o1),p(o1),f(o1),m(o1))
end rd

cl (o1,n)
       o1: 0
        n: C
       c(o1)<-n
end cl

cp (o1,r)
       o1: 0
        r: P->B
       p(o1)<-r
end cp

cr (o1,s1,z)
       o1: 0
       s1: S
        z: K
       make (o1)
       f(o1)(s1) <- {r,e,w,u,1}
       c(o1) <- 0
       p(o1) <- ∅
       k(o1) <- z
end cr
```

```
ds (o1)
     o1: 0
     f(o1)<-∅
     c(o1)<-0
     p(o1)<-∅
     break(o1)
end ds
```

3.3 Requirements of Military Security

3.3.1 Proofs of Correctness

The security state (1) of the system at any time i is described by the classifications, compartments and attributes of all the objects

$$q(i) = (c,p,f)$$

The system is initialized in some state q(o), (2) and by servicing updation commands evolves to security states q(1),q(2),...etc.

Given certain security criteria to be discussed below, our problem is to show that the system maintains these criteria. This entails two demonstrations

(i) Accession. Between changes in security state, i.e., while the system occupies security state q(i), the kernel enforces security requirements based upon privileges (and prohibitions) implied in q(i). (e.g., "no s can read o unless $f(s)(o) \geq \{r\}$").

(ii) Updation. In honoring a command and updating from q(i) to q(i+1), the kernel observes any updation constraints required by the performance criteria (e.g., "no subject may alter its own security classification".)

---

(1) This is identical to Lapadula and Bell's (8) notion of security state (p. 18) except for their component b.

(2) A typical q(0) would have one subject s0 the system administrator with full privileges to all system objects.

If (i) and (ii) can be demonstrated, then by induction on i, the system remains secure over time -- no sequence of access requests and updation commands can induce the kernel into a "security compromise." (1)

Before we can demonstrate (i) or (ii), we must delimit the criteria or rules which the kernel must enforce. Another way to say this is that we must define "security compromise".

It is here that debate will occur over what requirements to properly put upon the kernel. Based upon the tenet of "user responsibility" discussed above, we will list a reasonable set of rules demanded by military users. In section 3.3.2 we discuss the implications of our rules, and in section 3.3.3 we discuss possible alternatives.

The dichotomy (i),(ii) shown above breaks the criteria naturally into two parts - those regarding normal accession, and those regarding updation.

3.3.1.1 <u>Accession</u>.

Let q = (c,p,f) be a security state of the kernel. The rules are

(a) No s shall have any access to an o unless when access is requested

$$c(s) \geq c(o) \quad \text{and} \quad p(s) \geq p(o)$$

(b) No s shall be able to read, write on, execute, update the descriptor of or look at the descriptor of an object o unless

$$f(o)(s) \geq \{r\},\ \{w\},\ \{e\},$$

$$\{u\} \text{ or } \{l\}, \text{ respectively.}$$

(1) The notion of compromise, and the picture of the system as an automation evolving over time with command inputs, is due to Lapadula and Bell (8).

Proposition Provided

(i) all requests for access by subjects to objects are directed to the kernel

(ii) the kernel correctly retrieves and interprets the arguments of a request

(iii) the kernel correctly identifies the subjects and objects involved in a request

then

the system satisfies rules (a) and (b).

Proof. Consider the Access Evaluator program e. Subject s cannot access object o unless a system monitor performs the function for it. But e is interposed between all calls by s and the monitor. If (i), (ii) and (iii) hold, e blocks access of any kind unless $c(s) \geq c(o)$ and $p(s) \geq p(o)$, showing (a) holds. Given a request $b \in \{r,w,e,u,l\}$, access to $m(h(o,b))$ is blocked unless $f(o)(s) \geq \{b\}$, so (b) holds.

Q.E.D.

3.3.1.2 Updation

We list the updation constraints which should operate in a military environment

(c) No s may alter the descriptor of an object o unless $f(o)(s) \geq \{u\}$.

(d) No s may alter or read the descriptor of an object o unless $c(s) \geq c(o)$ and $p(s) \geq p(o)$.

(e) No access attributes may be granted by s1 to s2 for o unless

$$c(s2) \geq c(o) \quad \text{and} \quad p(s2) \geq p(o)$$

(f) No s may alter its own descriptor.

Proposition. Under the provisos (i) (ii) (iii) above and provided that in the initial security state q(0) we do not have $f(s)(s) \geq \{u\}$ for any s, then the system satisfies rules (c), (d), (e), and (f).

Proof. Consider the Update Constraint Checker program V. We take each rule in turn:

(c). Descriptors may only be altered via the updators wr, cl, cp, ds, The only calls to these functions occur from clauses preceded by an explicit check for $f(o)(s)\geq\{u\}$.

(d). Descriptors may only be altered or read by wr, cl, cp, ds, rd. Each is called from a clause which explicitly checks for $c(s)\geq c(o)$ and $p(s)\geq p(o)$.

(e). s1 can grant s2 attributes for o only by a call to wr(o,s2,-). This call occurs only in a clause preceded by the explicit check $c(s2)\geq c(o)$ and $p(s2)\geq p(o)$.

(f). s could alter its own descriptor only by calling wr, cl, cp, or ds on o=s, but each such call is preceded by an explicit check for $f(s)(s)\geq\{u\}$. Therefore if we can show that it is never possible to enter a security state with $f(s)(s)\geq\{u\}$ for any s, we are done. By hypothesis in q(0) we have no s with $f(s)(s)\geq\{u\}$. Suppose it were to occur in some q(i), and let i be the first such i. Then in q(i-1), not $f(s)(s)\geq\{u\}$. Hence V must have serviced a command at i resulting in wr(s,s,a1) with $a1\geq\{u\}$. But the call to wr(s2,o1,a1) is preceded by an explicit check not ($o1=s2$ and $a1\geq\{u\}$) which is violated by $o1=s2=s$ and $a1\geq\{u\}$. Thus we cannot have $f(s)(s)\geq\{u\}$ in q(i) or in any successor state of q(0).

(f) follows.

Q.E.D.

3.3.2 Implications. External Breaches.

In stating requirements (a) to (f) we have in effect defined the notion of internal security compromise - a compromise caused by the system's failure to meet responsibilities. Certain compromises of security in a larger sense can still occur through actions not under the control or scrutiny of the kernel. Examples of such external breaches are:

(1) A 3-cleared user s1 with r access to 3-classified file o1 copies o1 to o2, classifies o2 at 0 level, and grants read access to s2. User s2 is cleared only to 0. Even if the system could prevent direct

"moving" of files in such circumstances, s1 could still bypass the system by processing o1 into an altered form before copying to o2, could aggregate sensitive totals from o1 and copy them in o2, etc. No system could interpret all such possible evasions. Even if it could, s1 could still act by collusion as the direct agent of s2. Evidently, if s1 has privileged access to o1, no kernel can keep him from abuse of his trust.

An alternative to this approach is to force created files to be classified at the high watermark level of the environment of s1. Then either explicit declassification is prohibited, or, if not, this precaution is vacuous and at best a default convenience.

We choose to accept the axioms of complete trust in a priviliged user within the limits of his privileges and complete responsibility of the user in assigning classifications, compartments and attributes to files to which he has {u} privilege.

(2) A user s1 with clearance, compartments and {w} access for o1 can, even without {u} access, alter o1 beyond repair, in effect destroying it. There is therefore a good case for identifying the {w} and {u} attributes, merging them into a single {w} attribute. The design of e and V could be easily altered to accomodate this design decision, with essentially no changes in the arguments of section 3.1.

Another argument in favor of w=u is from user responsibility. If s1 can write in o1, s1 ought to be able to reclassify o1, since s1 may well have appended sensitive information to o1.

(3) A user s1 with u to o3 can provide another (suitably cleared) user s2 with any privileges to o3 he himself possesses, except s1 cannot grant {u} for s2 to s2. Prodigal use of this facility by s1 may result in an external breach, but the system cannot be responsible for making such distinctions.

3.3.3 <u>Alternative Kernel Designs</u>.

Certain other design possibilities can be handled with case in our framework. In each case they entail slight alterations of the e and V programs.

(1) Identifying $\{w\}$ with $\{u\}$. This was discussed in section 3.3.2.

(2) Allowing any subject s1 with $\{b\}$ attribute for o1, but without $\{u\}$ attribute, to <u>pass</u> $\{b\}$ to other subjects. This is similar to the 'transfer' ability of Graham and Denning (6). First we declare the function

$$U : (A\to B) \times (A\to B) \to (A\to B)$$

as the bitwise "or" of attribute lists. Then we alter the first conditional of V to read

```
if request='write' then
  begin
     if c(s1)≥c(o1) and p(s1)≥p(o1)
         and f(o1)(s1)≥{u} and
         c(s2)≥c(o1) and p(s2)≥p(o1)
         and not (o1=s2 and a1≥{u})
    then  wr (s2,o1,a1)
    else  if c(s1)≥c(o1) and p(s1)≥p(o1)
         and f(o1)(s1)≥a1 and c(s2)≥c(o1)
         and p(s2)≥p(o1)
    then
         begin
              a1<-a1 U f(o1)(s2)
              wr(s2,o1,a1)
         end
    else m(0)
   end
else if request = 'read' then ...
```

(3) Allowing more limited updation privileges than those implied by $\{u\}$. Thus $f(o1)(s1)\geq\{n\}$ might allow s1 to change only access attributes to, but not clearance or classification of, object o1 while $f(o1)(s1)\geq\{j\}$ would be needed to reclassify.

(4) Enforcing a requirement that each object o1 have an unique 'owner' (Graham and Denning (6), p. 420.). We can capture this idea by allowing only one s1 to have $\{u\}$ to o1. Assuming this is the case in the initial security state q(0), we build into V the check

```
if request = 'write' then
   begin
```

```
    if c(s1)$\geq$c(o1) and p(s1)$\geq$p(o1)
         and f(o1)(s1)$\geq\{u\}$ and
         c(s2)$\geq$c(o1) and p(s2)$\geq$p(o1)
         and  not  a1$\geq\{u\}$
    then  wr (s2,o1,a1)
    else  m(0)
  end
```

Then an inductive argument shows that, since $\{u\}$ can never be "passed", no o1 ever has more than one s with f(o1)(s)$\geq\{u\}$. Since every created object has a default "owner" (its creator), the uniqueness requirement is proved.

(5) In the view of Burke (2) access privileges granted to s1 for o1 should depend upon the mode m(o1). For example it is meaningless to grant $\{c\}$ access to a data file. Thus he proposes that the kernel at update time check m(o1) and grant only the appropriate attributes.

By adding further conditionals to the updators we can accomodate this constraint. For example wr may be altered to

```
wr (s2,o1,a1)
     if m(o1)=p
        then f(o1)(s2)<- a1$\cap\{e,u,1\}$
        else if m(o1)=f
            then f(o1)(s2)<-a1$\cap\{r,w,u,1\}$
               ... etc
```

REFERENCES

1. AFR 205-1. Safeguarding Classified Information. Dept of the Air Force, 2 Jan 68.

2. Burke, E. L., Private Communication

3. DoD 5200.28-M. Manual of Techniques and Procedures for Implementing, Deactivating, Testing and Evaluating Secure Resource-Sharing ADP Systems, 15 Aug 72.

4. Goheen, S. M. and R. S. Fiske. OS/360 Computer Security Penetration Example. MITRE WP-4467, The MITRE Corporation, 16 Oct 72.

5. Graham, G. S. Protection Structures in Operating Systems. Master's Thesis, Dept of Computer Science, Univ of Toronto, Aug 71.

6. Graham, G. S. and P. J. Denning. Protection-Principles and Practice. AFIPS Proc Spring Joint Computer Conf (1972), 417-429.

7. Lampson, B. W. Protection. Proc 5th Annual Princeton Conf on Information Sciences and Systems. Princeton Dept of Elec Engineering (March 1971), 437-443.

8. La Padula, L. J. and D. E. Bell. Secure Computer Systems: Mathematical Foundations. Draft MTR, The MITRE Corporation, 2 Oct 72.

9. Popek, G. J. On the Design of Secure Systems. elsewhere in this volume.

10. Schell, R. R. Notes on an Approach for Design of Secure Military ADP Systems, elsewhere in this volume.

11. C. Weissman. Security controls in the ADEPT-50 time-sharing System. Proc FJCC (1969), 119-133.