ADA039324

INTEGRITY CONSIDERATIONS

FOR SECURE COMPUTER SYSTEMS

APRIL 1977

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

D D C
MAY 12 1977
A

AD No.
DDC FILE COPY

ESD-TR-76-372

MTR-3153
Rev. 1

INTEGRITY CONSIDERATIONS

FOR SECURE COMPUTER SYSTEMS

APRIL 1977

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

D D C

MAY 12 1977

A

Project No. 522B
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-75-C-0001

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

WILLIAM R. PRICE, Captain, USAF
Techniques Engineering Division

ROGER R. SCHELL, Lt Colonel, USAF
AF System Security Program Manager

FOR THE COMMANDER

STANLEY P. DERESKA, Colonel, USAF
Deputy Director, Computer Systems Engineering
Deputy for Command and Management Systems

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
ESD-TR-76-372		

4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED
INTEGRITY CONSIDERATIONS FOR SECURE COMPUTER SYSTEMS.	
	6. PERFORMING ORG. REPORT NUMBER
	MTR-3153-Rev-1

7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
K. J. Biba	F19628-75-C-0001

9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
The MITRE Corporation Box 208 Bedford, MA 01730	Project No. 522B

11. CONTROLLING OFFICE NAME AND ADDRESS	
Deputy for Command and Management Systems Electronic Systems Division, AFSC Hanscom Air Force Base, Bedford, MA 01731	13. REPORT DATE APRIL 1977
	13. NUMBER OF PAGES 66 p.

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

COMPUTER MODEL	INTEGRITY MODEL
COMPUTER SECURITY	PROGRAM INTEGRITY
DATA INTEGRITY	
INTEGRITY MECHANISMS	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An integrity policy defines formal access constraints which, if effectively enforced, protect data from improper modification. We identify the integrity problems posed by a secure military computer utility. Integrity policies addressing these problems are developed and their effectiveness evaluated. A prototype secure computer utility, Multics, is then used as a testbed for the application of the developed access controls.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73

ACKNOWLEDGMENTS

1

TABLE OF CONTENTS

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

SECTION I

INTRODUCTION

OVERVIEW

Ensuring legitimate access to privileged information has become a major area of concern for information processing technology. The rapidly growing use of complex resource sharing information systems[1] has emphasized the need to carefully identify and guarantee who has which access to what data. Experience has indicated that the protection issue is two-pronged: concerned both with the proper dissemination of information and with that information's validity. Our concern, in this paper, is an examination of how information validity may be maintained.

Our context is the Secure General Purpose Computer Project of the Air Force's Electronic Systems Division [1]. Its purpose is the design, construction, and formal validation of a secure computer utility for the military environment. The term secure computer utility refers to an interactive, multiprogrammed, multiprocessor computer system supporting resource sharing in a manner determined by an information protection policy. Its effective enforcement of the protection policy must be formally validated before it can be certified, by the appropriate authority, for use with classified information.

The protection policy of particular interest in a military environment [2] [3] takes the form of the DoD security regulations. To date, security has been interpreted to address the authorized observation (dissemination) of classified information. We intend to extend the certified function of the utility to address the proper modification of information within the computer system.

---

[1] These systems are realized in many forms including: network packet-switches, time-shared mainframes, microcomputer arrays, and dedicated data base systems.

BACKGROUND

Before beginning a discussion of information validity, a re-
view of fundamental notions upon which this paper (and the project)
is built is appropriate. These are:

>   1) the reference monitor;
>
>   2) a formalized security policy; and
>
>   3) the concept of a kernel.

Those readers familiar with these concepts may skip ahead to
Section II where we begin our analysis of an integrity policy.

The Reference Monitor

An investigation [4] into the protection problems of military
computer systems proposed the abstract notion of a reference
monitor as a generic solution.
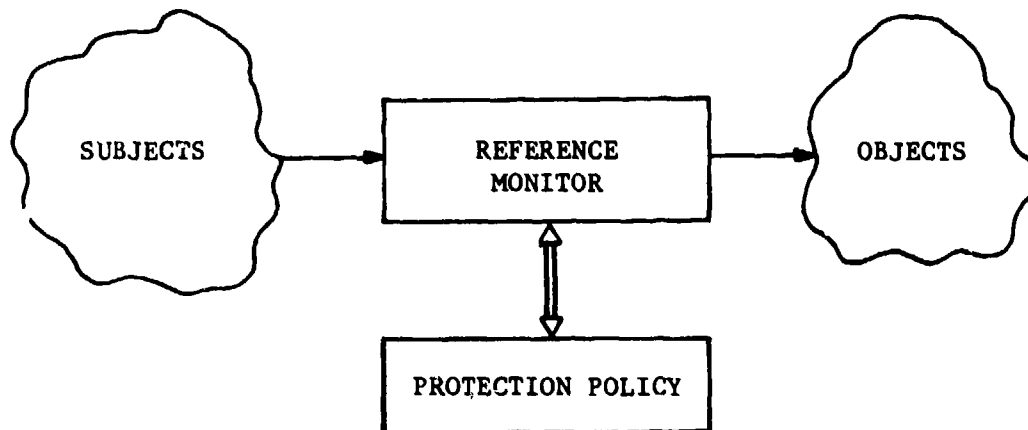
Figure 1. A Reference Monitor

A reference monitor is an entity that monitors and decides the allowability of all accesses by information processors (subjects) to information repositories (objects). The protection policy, enforced by the reference monitor, is reflected in the data base in which the allowability decision is encoded.

A reference monitor must satisfy three logical properties:

1) it is complete: <u>all</u> accesses by subjects to objects are monitored and enforced;

2) it is protected: its function may not be maliciously or accidently modified by unauthorized forces; and

3) it has provably proper behavior: it must faithfully enforce the specified protection policy.

We will see that the second property is concerned with the maintenance of the validity of the reference monitor.

The protection policy has historically [2] [3] [5] been represented as a set of axioms constraining access by subjects to objects. A generic policy model will illustrate the concept. We define a set S of subjects, a set O of objects, a relation $\underline{a} \subseteq S \times O$ denoting that a subject s may access an object o, and a function <u>domain</u>: $S \rightarrow POWERSET(O)$. The axiom A1.1 defines the function of the reference monitor.

(A1.1) $\forall s \in S, o \in O$ s $\underline{a}$ o $\Rightarrow$ o $\in$ <u>domain</u>(s).

The function <u>domain</u> is an encoding of a protection policy identifying the accessible name space of its subject argument. A1.1 specifies that a subject s may access an object o only if o is in the domain of s. A protection policy takes the form of a decision algorithm defining the name spaces (or partitions of name spaces) assigned to subjects. The dotted lines of Figure 2 represent the access domains of the two subjects S1 and S2. The arrows have the following interpretation: arrows pointing to an object represent a modification of an object by a subject; arrows pointing to a subject from an object represent an observation of the object by the subject.

7

Access Modes

The above example presented a rather abstract form of access, independent of the semantics associated with an access. In general, access domains are partitioned on the basis of the mode (semantics) of the access. In this paper we will be concerned with three abstract modes of access: observation, modification, and invocation. Access permission, for each of these modes, will be explicitly identified.

Observation, as the name implies, relates to the viewing of information by a subject. Central to observation is the testing of information. We state that observation is the testing of information that results in a choice of distinct states of the observing subject (and possibly distinct outputs). In other words, the observing subject can make a choice based on the observed information, and that choice manifests itself in the resulting state of the observer.

Modification may be defined in terms of observation. A subject modifies information if its value is changed so that an observation, by a subject (possibly distinct from the modifier), results in a different state than previous observations (a discernable change).

Invocation is a logical request for service from one subject to another. Since the control state of the invoked subject is a function of the fact that the subject was invoked, invocation is a special case of modification. Invocation is an abstraction of the primary control construct for transferring control between distinct subjects in (possibly) differing access domains. In general we require that the invoking subject not be informed of the success or failure of the operation. Such information may be passed by subsequent invocations, the sequencing determined by some control protocol. Interprocess communication is one instance of an invocation mechanism where the wakeup signal passed from the invoking process to the invoked process constitutes the invocation. A subroutine type of intersubject control tructure may be accomplished by two invocations: the invoker first signals the invoked subject and then waits for the invoked subject to reactivate it (return) via a subsequent invocation. A necessary consequence of the subroutine type of control structure is that the invoking and invoked subjects must each have invocation privilege to the other

It should be noted that "execute" access is quite different than invocation. While invocation represents a control access between distinct subjects in (possibly) differing domains, execution is the access, by a subject, to an object for the purpose of obtaining

8

instructions. Since during execution a subject obtains its
instructions by observing the object in which the instructions
reside, we will consider execute access to be equivalent to observe
access for our purposes.

From these elementary constructs, we may compose complex sub-
systems accessing many objects (subjects) with combinations of pri-
mitive access modes.

Security Policy

The protection policies invest'gated, to date, have addressed
the problem of information security. Security denotes the property
of protection against compromise: unauthorized dissemination of
information. The security policy defines access domains of subjects
based on considerations derived from DoD security attributes of sub-
jects and objects. Several axiomatic systems [2] [3] represent this
policy. We present a model defining this policy below.
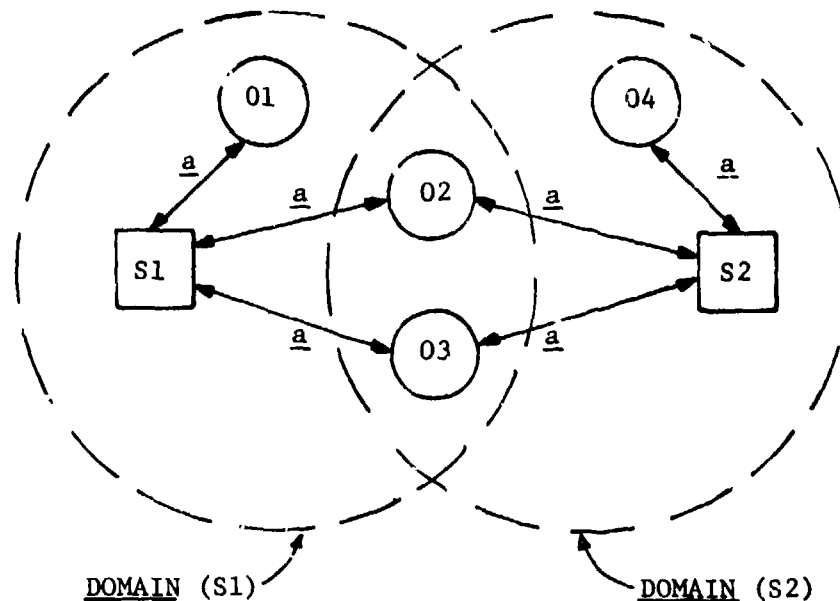
Figure 2. Access Domains

First, we define the elements of the model.

S: a set of subjects;

O: a set of objects where the intersection of S and O is null;

SL: a partially ordered set of security levels that forms a
     lattice;

$\underline{sl}$: S x O -> SL, a function mapping subjects and objects into
     security levels;

$\underline{\leq}$: a subset of SL x SL defining the partial ordering "less
     than or equal";

$\underline{o}$: a subset of S x O defining the access capability for
     observation; and

$\underline{m}$: a subset of S x O defining the access capability for modi-
     fication.

The access domain of each subject is partitioned with respect
to the mode of access. The following axioms[2] define the access
domain.

(A1.2) $\forall s \in S, o \in O$ $s \underline{o} o \Rightarrow \underline{sl}(o) \leq \underline{sl}(s)$.

(A1.3) $\forall s \in S, o \in O$ $s \underline{m} o \Rightarrow \underline{sl}(s) \leq \underline{sl}(o)$.

The access domain for each subject s is thus the set:

$$\left\{ o \in O \mid \underline{sl}(o) \leq \underline{sl}(s) \quad \underline{or} \quad \underline{sl}(s) \leq \underline{sl}(o) \right\}.$$

A subject may observe the information contained in an object if its
security level is greater than or equal to that of the object. A
subject may modify the information contained in an object if its se-
curity level is less than or equal to that of the object. These
constraints insure [2] [3] that information may be transferred only
"upward" in security level, even by subjects untrusted to behave
properly.

We may also extend the above axioms to intersubject invocation
by the relation $\underline{i}$: subset of S x S defining the access capability
for intersubject invocation.

_____

[2]Formal statements are labelled according to the following conven-
tion: a section specific number prefaced by "A" for axioms, "T"
for theorems, and "D" for definitions.

(A1.4) $\forall$ s[1],s[2] $\in$ S   s[1] $\underline{i}$ s[2] => $\underline{sl}$(s[1]) $\leq$ $\underline{sl}$(s[2]).

The Kernel Concept

The realization of a reference monitor within a computer system is termed a kernel. Conceptually, the kernel is a central, localized hardware/software system that enforces the protection policy of the reference monitor it implements [1]. The kernel defines an abstract machine composed of logical objects and operations[3], access to which is determined by the protection policy. A crucial part of the kernel development process is the formal verification of the property: the protection policy is enforced for all designated accesses.

The first kernel in the ESD program was constructed for the PDP-11/45 [6]. Its successful implementation encouraged the design and implementation of a large-scale prototype, based on the Honeywell Information Systems' Multics [7]. The kernel design for this system incorporates the protection considerations described in this paper.

OUTLINE

We will address the following issues:

1) identification of relevant integrity problems;

2) definition of protection policies that address these problems; and

3) identification of the computer system elements (subsystems) to which these protection policies should be applied.

Our analysis of integrity policies begin in Section II with a discussion of integrity problems within a military computer utility. Section III formally proposes several integrity policies designed to cope with the problems posed in Section II. Section IV considers the application of these policies within a prototype computer utility.

---

[3]Implemented either in software, firmware, or hardware.

11

A caution to the reader. This investigation occurs in the context of a Multics' kernel development. Therefore many of the examples (and particularly the jargon) are taken from this milieu. While the issues (and conclusions) generalize, some familiarity with Multics [7] is useful.

SECTION II

THE INTEGRITY PROBLEM

INTEGRITY DEFINED

What do we mean by integrity in computer systems? In this sub-- section we will investigate our intend*d* meaning of integrity and establish the context for the integrity formulations of the succeed- ing sections.

Webster's dictionary provides an initial integrity definition:

integrity - 1a: an unimpaired or unmarred condition: entire
    correspondence with an original condition: SOUNDNESS
    1b: an uncompromising adherence to a c*u*de of moral,
    artistic or other values.

This definition, and our informal notions of integrity point to a similar conception: integrity does not imply guarantees concern- ing the absolute behavior of systems. For instance, a p*e*rson thought to have the property of integrity is only considered to behave consistently with respect to some standard: no statement (or decision) about the quality of the standard is implied.

This concept can be applied to computer systems. We consider a subsystem to possess the property of integrity if it can be trusted to adhere to a well-defined code of behavior. No a priori statement as to the properties of this behavior are relevant.

The concern of computer system integrity is thus the guarantee that a subsystem will pe--form as it was intended to perform by its creator. We assume that a subsystem has been initially determined (by some system external agency) to perform properly. Program verification technology addresses just this problem of initial sub- systems validation. We then wish to insure that the subsystem cannot be corrupted to perform in a manner contrary to the original determi- nation. The integrity problem is the formulation of access control policies and mechanisms that provide a subsystem with the isolation necessary for protection from subversion. Based on an initial assumption of proper behavior (according to some system external standard), we are primarily concerned with protection from inten- tionally malicious attack: unprivileged, intentionally malicious modification.

INTEGRITY THREATS

How can integrity be compromised? That is, how can a system be improperly persuaded (forced) to change its behavior? The following paragraphs briefly classify integrity threats. The abstract form of an integrity threat is a subsystem modification not considered in the subsystem's initial verification of proper behavior. We term such improper modifications sabotage.

Our viewpoint is that of the subsystem: some subset of a system's subjects and objects isolated on the basis of function or privilege. We will consider two dimensions of integrity threats to a subsystem: source and type. Integrity threat sources identify where a threat might originate while threat type identifies the manner in which the threat might be made.

Threat Sources

We consider two threat sources:

1) subsystem external; and

2) subsystem internal.

Their names unambiguously describe their origin. An external threat is posed by one subsystem attempting to change (improperly) the behavior of another by the supplying of false data, improperly invoking functions, or direct modification of its own behavior. Improperly performed (or not considered in the specification of its behavior) modification of behavior can sabotage subsystem function. Internal threats arise if the subsystem is malicious or, more likely, incorrect. This threat is addressed by program verification techniques.

Threat Types

A second classification of threats can be made on the basis of type. We consider two:

1) direct (overt); and

2) indirect (covert).

Direct threats involve "direct" means: a write into a protected data base object. We must consider, in this case, the protection properties only of the accessing subject and accessed object. Indirect threats subsume a much larger class of scenarios. Generally, indirect threats refer to improper modifications resulting from the use of data or procedures developed (modified) by a malicious subsystem. This data (procedure), by not fulfilling expected

14

requirements, may then sabotage its user's functions. This struc-
ture requires knowledge, at each access, of both the ultimate source
and transfer path of the accessed information.

Examples

We can illustrate this threat taxonomy by an example drawn from
the "people system." Let us consider threats to the physical inte-
grity of a person. External threats take the form of another person
perpetrating physical harm. Internal threats take the form of self-
inflicted physical harm. For instance:

- external direct: a direct assault by another person with,
  say, a knife;

- external indirect: an assault by another person via surrepti-
  tious means, say, by poison covertly placed in food;

- internal direct: suicide via direct means, say, a gun; and

- internal indirect: unsuspecting suicide via, say, poor care
  for one's body.

A protection policy can be formulated to block each of these
threats. For example, a policy which identifies persons who might
commit assault can be used to segregate them so that they have no op-
portunity (no access) to commit assault and to insure that they are
not hired as cooks or servers. In these cases, selective isolation
is sufficient protection. However, what about internal threats?
These can only be blocked by some certification of an individual's
ability and desire to take proper care of himself.

The example can be easily extended to computer systems. For
instance:

- external direct: one subsystem maliciously modifies a
  necessary data base, say executable code, of another sub-
  system;

- external indirect: one subsystem foists a maliciously
  behaving subroutine onto another subsystem (Trojan Horse
  attack);

- internal direct: self-modifying code; and

- internal indirect: inadvertent self-modifying code, say,
  via improperly initialized pointers.

External and internal threats are illustrated in Figure 3. Subject
S1 maliciously modifies object O1, causing subject S2's behavior to

15

change. Subject S3 modifies its own data base (object O2) in an
unanticipated manner, causing changes in its subsequent behavior.
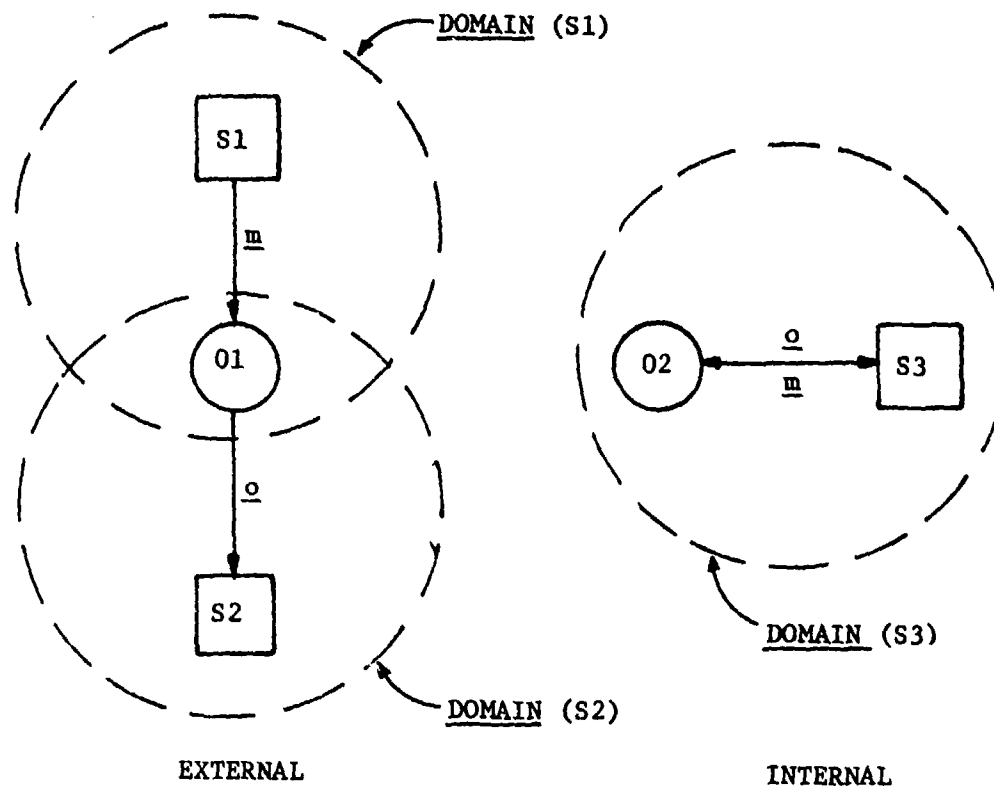
Figure 3. External and Internal Integrity Threats

INTEGRITY POLICY ENFORCEMENT

In each of the above cases, our primary concern is an identifi-
cation of those modifications which preserve the validity (intended
properties) of computer system elements. Integrity policy concisely
organizes this information so that the propriety of a given access
can be easily evaluated by an enforcement mechanism. Two issues
govern the types of integrity policy a given system may support:

1) available enforcement mechanisms; and

2)   the integrity problems the system must address.

This subsection will address the former issue, the following subsection the latter.

The space of enforcement mechanisms is diagrammed in Figure 4. We identify two dimensions: enforcement frequency and enforcement granularity. Frequency refers to the time at which access control enforcement occurs. If enforcement is performed only once (viz., at the time a program is created) the frequency is termed static. If enforcement is performed at each access by a program to an object, the frequency is termed dynamic. One-time program verification of access propriety is a static enforcement mechanism. Hardware descriptor mechanisms support dynamic enforcement.

ENFORCEMENT FREQUENCY

STATIC

COARSE   FINE   ENFORCEMENT
                                           GRANULARITY

DYNAMIC
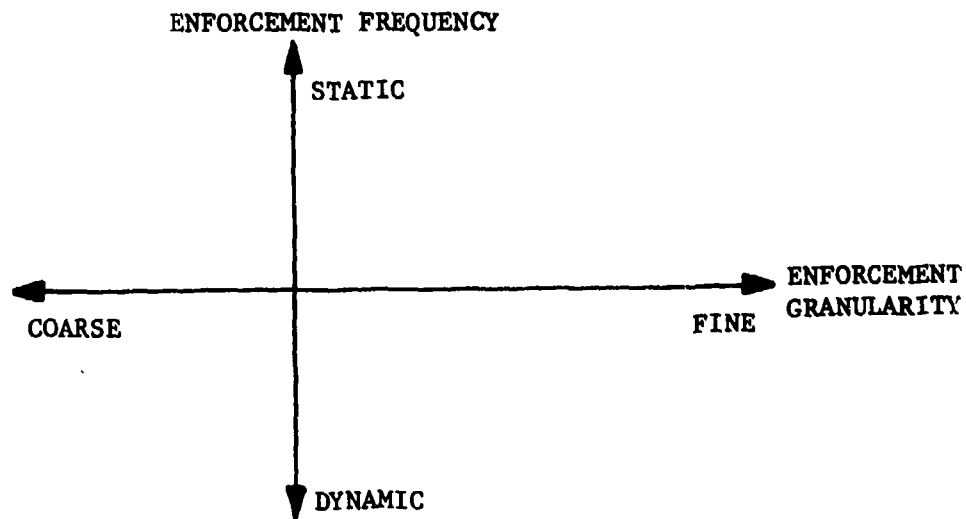
Figure 4.   Enforcement Mechanisms

Granularity refers to the size and resolution of the protected system elements. For effective enforcement of an integrity policy, the granularity of enforcement must match the granularity of the policy. For example, if a protection policy controls access to parts of a file, an enforcement mechanism that only controls access to the entire file cannot effectively implement the policy.

Another example mechanism supports static enforcement at a coarse granularity. The propriety of access by a program to coarsely defined objects (entire files) may be statically evaluated by verification of the program text at the time the program was created. We can insure, via this technique, that the program only accesses certain named files in a proper manner. However, if the renaming of files is a facility supported by the system, this verification can be invalidated by the renaming of files accessed by the program. We note that the solution to this renaming problem centers about the matching of the frequency and granularity of access enforcement with the frequency and granularity of the binding of program names to objects.

These examples illustrate that policies that alter the protection attributes and existence of subjects and objects must have dynamic enforcement at the granularity of the attribute alteration to support effective access control. Since a computer utility (our intended application) supports a dynamic environment with these properties, our subsequent discussions focus on integrity policies suited for a dynamic environment.

The integrity of information is maintained by guaranteeing that only proper modifications are made. As indicated above, this can be done in a number of ways ranging from access control at execution time to program verification. We find that internal threats, in general, cannot be addressed by dynamic access control mechanism. Indeed, we note that for any given computer system, the hardware access control mechanism defines a certain level of subsystem access control granularity. Any access restriction below this level of granularity (or based on other than hardware defined access modes) must be guaranteed by static enforcement.

The static nature of internal threat policy enforcement places it beyond the scope of this report. The prevention of internal threats is more the province of program verification. Therefore, our concern focuses on integrity policies for external threats.

INTEGRITY PROBLEMS

Specific interpretations of the notion of "proper" modification and the consequent protection policies are dependent on problem specific protection requirements. This subsection does not address all conceivable protection problems: an obviously impossible task. It does identify certain problems that have, historically, proved troublesome.

We begin by restating the fundamental principle of integrity policy: identification and enforcement of proper modifications. Our job, then, must start with the isolation of relevant, proper modifications. For a secure DoD computer utility, three classes of proper modifications are of immediate interest:

1) propriety of modification with respect to the national security importance of information;

2) propriety of modification with respect to the identity of an accessing user ("need-to-modify"); and

3) propriety of modification with respect to application specific reasons.

We consider each in turn.

National Security

The integrity of national security information is clearly of paramount importance for the intended user community. The malicious modification of crucial information can, in some circumstances, be of greater importance than its compromise (unauthorized observation). Consider the case of global data bases accessible to a large community of innocuous applications yet of critical importance to a few applications. For example, a data base defining interstate transportation routes is useful to a large number of non-critical tasks. Yet the sound construction of this data base is crucial to the proper operation of logistics programs in time of national emergency. Clearly, this data base must be protected from modification to a degree commensurate with its importance to national security (its accessing applications), while still being observable to a variety of applications at differing security levels.

A similar situation pertains to access to procedures. Consider the ill-conceived use of a library subroutine by a critically important program. An optimal routing subroutine of a logistics management package is a good example. If the subroutine is not protected from malicious modification at least to the degree of the subject using it, sabotage of its function may cause improper behavior of its caller. Yet the library routine should be executable (observable) by the entire user community, at many levels of importance to national security.

Thus, a contradiction exists between the protection necessary for observation and modification. A single security level cannot be assigned to these objects so as to satisfy both protection requirements: (observable by everyone) and (modifiable by no one). This

19

apparent difficulty must be addressed by an integrity policy distinct from the security (compromise) policy.

Available DoD policy [8] requires access controls that guarantee data integrity: that is, the accessibility, maintenance, movement, and disposition of data shall be governed on the basis of security classification and need-to-know. However, the interpretation of such controls has centered on the security issue: information compromise. Our requirement is the specification of protection policies addressing the sabotage of information important to the national security.

We should note that similar considerations apply to applications other than the military. The need to compartmentalize data modification exists in a variety of application areas.

User Identity

Protection against improper modification based on user identity takes a somewhat different form. Consider a data base owned by user A. User B wishes to read this data base. User A does not trust user B to non-maliciously access an only py and would like to extend read-only access to this specific data base only to user B and his subjects (from the set of all users). If user B misuses his privilege to the data base, user A must have the capability to revoke user B's access. The protection mechanism must support dynamic revocation based on user identity. It should be noted that while the previous problem isolates classes of users (by national security privilege) it is orthogonal with respect to this problem since a given user may operate at many levels of national security privilege.

Protected Subsystems

The preceding problems addressed access restrictions based on properties of the person who either invokes or creates a subsystem. A distinct class of protection problems arise from the use of system services. These services take the form of subjects, system-supplied, that may be invoked by user-supplied subjects to perform privileged functions. These subjects (and the data and procedure they access) must be explicitly protected from their invoker. Likewise, the invoking subject often requires protection from the invoked subject.[4]

---

[4]This situation has been referred to as the "mutually suspicious problem" [9].

20

Two important instances of protected system services are:

1) a security kernel; and

2) a distributed operating system.

However, in these instances the security kernel and operating system require access to user defined data bases (viz., parameters). Thus, in the simple case, the operating system has unlimited access to user space, but the user has only carefully restricted access to operating system space.[5] Our consideration of this problem will not be general. Rather, we concentrate on explicating a policy sufficient to isolate a kernel.

All three of the above examples of specific problems are instances of the archetypal model presented in Section I. Each policy attempts to isolate domains of access privilege, each (possibly) disjoint from those defined by the other policies. The access frequency and general properties of each suggest different implementation mechanisms. Indeed, the fundamental property of rate of change of privilege, either by the accessing subject or by accessed object motivates the tailoring of policy and mechanism to the properties of the access.

PROTECTION ENVIRONMENTS

Subsystems within a computer system perform differing functions and thus often require differing integrity policies. The protection requirements are also a function of the perspective with which subsystems are viewed. We make the following definitions of perspective:

1) each system user has (at least) one process executing in his/her behalf;

2) each process is composed of a number of subjects and objects, partitioned into domains of access privilege within each process; and

3) some subset of the most privileged subjects and objects within each process define the protected subsystem that comprises the security kernel.

_____

[5] Primarily through "system service calls" (viz., "gates" in Multics) that permit controlled invocation of operating system functions by the user. An important special case is the invocation of the kernel from uncertified user subsystems.

Based on these definitions, we can identify two environments with distinct integrity requirements:

1) the set of subjects and objects not contained in the security kernel. These will be collectively referred to as the user virtual environment; and

2) the set of subjects and objects that compose the security kernel. These will be collectively referred to as the kernel environment.

Section IV applies the integrity policies developed in Section III to each of these protection environments.

SECTION III

INTEGRITY POLICY

TYPES OF POLICY

Two classes of integrity policy are important: mandatory and discretionary. They differ in the manner in which protection policy, once made, may be changed. Mandatory integrity policy refers to a protection policy which, once defined for an object, is unchangeable and must be satisfied for all states of the system (as long as the object exists). The static nature of the policy is governed by (system) external controls. In our case, the policy addressing protection of national security information will be of a mandatory nature. The decision as to the importance to national security of the integrity protected information is not one that the system may make internally. On the other hand, a discretionary policy is one in which the protection policy may be dynamically (during the existence of an object) defined by the user. The latter two examples of Section II illustrated instances of discretionary integrity policies.

The following discussions will, in turn, discuss mandatory and discretionary modification protection policies for secure DoD Multics.

MANDATORY INTEGRITY POLICY

A mandatory protection policy addressing the integrity of national security information must consider two issues: 1) the identification of protected abstract objects, and 2) the determination of permissible modification access. The following analyses define a set of primitive models which, while sharing common abstract object identification, propose differing policies for constraining access.

The Elements of Policy

Our presentation of mandatory integrity policies follows a style similar to that of [3]. Each policy is defined as a set of relations, one for each type of access mode defined, on the sets of active and passive computer system elements. This latter categorization identifies those system elements which perform information accesses (subjects) and those which are accessed (objects). Clearly,

23

differing protection policies may protect differing kinds of objects from differing kinds of subject's accesses. The identification of subjects and objects is thus often policy dependent. The terms agent and repository identify analogous elements in [3]. A protection policy is identified by a decision rule which determines the access relations.

It is the decision rule which embodies the content of the policy. With respect to national security information,[6] these decision rules, for varied policies, have certain common features. The governmental security system uses the mechanism of a security level to define the trustworthiness of an individual with respect to national security information. While only properly addressing information compromise, it is apparent that the trustworthiness of an individual, as formalized through his security level, also addresses information sabotage.[7] Thus each individual's security level can serve to delimit a range of integrity levels for which the individual, or his/her program, is trusted not to perform malicious modification. The assignment of an integrity level to a subject is determined by two constraints: 1) the permitted integrity level range of the associated user, and 2) the minimum[8] integrity level necessary to usefully access protected information, required for the subject's intended function.

The integrity level or importance assigned to objects may be determined in a manner directly analogous to that used for security level assignment. A security level is assigned to information (and thus objects) on the basis of possible national security damage caused by disclosure. An integrity level is assigned on the basis of possible national security damage caused by information sabotage.

The set of integrity levels is defined by the product of the set of integrity classes and the powerset of the set of integrity compartments. The resultant set forms a lattice under the partial ordering leq, similar to the relation $\leq$ defined for security levels [3].

---

[6] Indeed, any common protection problem domain.

[7] An individual trusted not to divulge information is not likely to maliciously modify it.

[8] The concept of the assignment of the least access privilege [10]. necessary to accomplish a task is applicable.

24

The set of integrity classes may be disjoint from the set of security classes. This result is not too surprising since they are assigned, and used, for different purposes. However, these sets will have common properties. Indeed, a set of integrity classes may be defined as:

-TOP SECRET: information whose unauthorized modification could reasonably be expected to cause exceptionally grave damage to our national security;

-SECRET: information whose unauthorized modification could reasonably be expected to cause serious damage to our national security; and

-CONFIDENTIAL: information whose unauthorized modification could reasonably be expected to cause damage to our national security.

The above prototype integrity classes were chosen to correspond to the security classes. This correspondence is suggested because of the strong analogy between security and integrity.

The set of integrity compartments serves much the same purpose as the set of security compartments: to partition the sets of subjects and objects on the basis of functional area. For example, some integrity compartments might identify differing applications: logistics, simulation, real-time command and control, or budget control. An integrity level for a computer system element is composed of an integrity class, identifying the national security importance of the element, and a set of integrity compartments identifying the information partitions the element may contain or access.

The above discussion illustrates three points. First, the assignment of integrity levels to individuals is based on very similar considerations to the assignment of security levels: the trustworthiness of individuals. It is not very practical to partition the trustworthiness of individuals with respect to disclosure and sabotage. Human behavior cannot be so neatly divided. Thus, similar considerations apply in the assignment of both security and integrity levels for individuals. A subject is assigned an integrity level commensurate with the level of its user and with the principle of least privilege.

Second, the assignment of integrity levels to objects is based on quite different criteria than the assignment of security levels. The example of the transportation data base in Section II illustrates this point. Integrity levels, by definition, are assigned not to prevent information disclosure, rather to prevent information sabotage.

Third, the issue of whether the same values (viz., classes and formal compartments) are used for the set of integrity levels and the set of security levels is not decided.

For our examples, we will use the above set of integrity levels, respectively labelled TS = TOP SECRET, S = SECRET, and C = CONFIDENTIAL. They are, like security levels, partially ordered: C _leq_ S _leq_ TS.

Definitions

Each model to be presented has the following basic elements:

S: the set of subjects s, the active, information process-
       ing elements of a computing system;

O: the set of objects o, the passive information repository
       elements of a computing system (the intersection of S
       and O is the null set);

I: the set of integrity levels discussed above;

_il_: S x O -> I; a function defining the integrity level of
       each subject and object; defines a lattice under the
       relation _leq_;

leq: a relation (subset of I x I) defining a partial ordering
       "less than or equal" on the set of integrity levels I;

less: an antisymmetric, transitive relation (subset of I x I)
       defining the "less than" relationship on the set of
       integrity levels I;

min: POWERSET(I) -> I, a function returning the greatest
       lower bound (meet) of the subset of I specified;

26

o: a relation (subset of S x O) defining the capability of a subject, s $\in$ S, to observe an object, o $\in$ O: s o o;

m: a relation (subset of S x O) defining the capability of a subject, S $\in$ S, to modify an object, o $\in$ O: s m o;

i: a relation (subset of S x S) defining the capability of a subject, s[1] $\in$ S, to invoke another subject, s[2] $\in$ S: s[1] i s[2]. This operation can be considered a prototype for interprocess communication and procedure call.

We now describe three alternative mandatory integrity policies. The policies are defined by axioms constraining the elements of the sets o, m, and i. The described models do not exhaustively describe all possible policies. However, they identify a representative set of useful models.

The Low-Water Mark Policy

The concept of a security high-water mark found early application in secure computer systems [11]. The high-water mark security policy relates current access privilege of a subject to the highest security level possessed by observed objects.[9] Its fundamental concept may be equally applied to an integrity policy. Unlike subsequent mandatory models we will examine, the low-water mark model is dynamic, in the sense that the integrity level of a subject is not static, but is a function of its previous behavior.

The policy provides for a dynamic, monotone and non-increasing value of $il(s)$ for each subject. The value of $il(s)$, at any time, reflects the low-water mark of the previous behavior of the subject. The low-water mark is the least integrity level of an object accessed for observation by the subject. Further, a subject is constrained to modify only those objects which possess an integrity level less than or equal to the sui   ,. These properties are formalized by the following axioms.

---

[9] Our interpretation of the policy differs substantially in that we do not permit modify access to objects having a lesser (or incomparable) security level than the accessing subject. A valid criticism of the policy in [11] relates to an inherent vulnerability to indirect threats (Trojan Horse attacks).

(A3.1) For each observe access, of an object o by a subject s, the integrity level of the subject, $\underline{i1}'(s)$, immediately subsequent to the access, is defined by:

$$\underline{i1}'(s) = \underline{min} \left| \underline{i1}(s), \underline{i1}(c) \right|$$

where $i1(s)$ is the integrity level of s immediately preceding the access.

(A3.2) $\forall s \in S$, oθ0 s $\underline{m}$ o $\Rightarrow$ $\underline{i1}(o)$ $\underline{leq}$ $\underline{i1}(s)$.

(A3.3) $\forall s[1], s[2] \in S$ $s[1]$ $\underline{i}$ $s[2]$ $\Rightarrow$ $\underline{i1}(s[2])$ $\underline{leq}$ $\underline{i1}(s[1])$.

A3.2, by construction, insures that direct malicious modification is impossible. Satisfaction of A3.1 insures the indirect sabotage, by use of "contaminated" data or procedure, is also impossible.

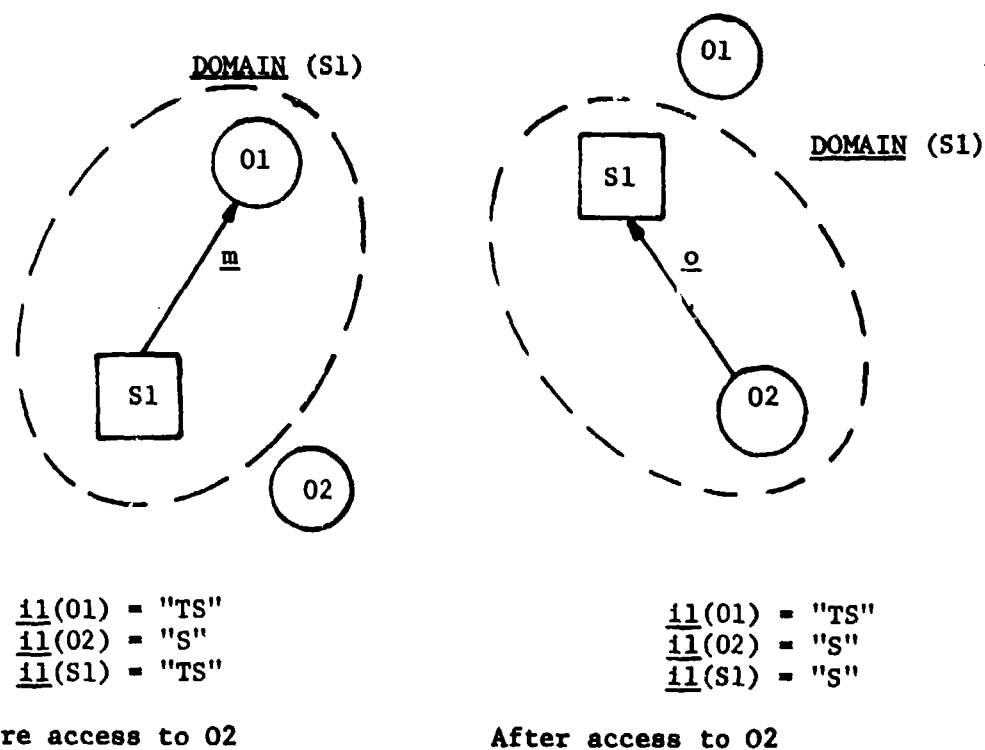$\underline{i1}(01) = $ "TS"
$\underline{i1}(02) = $ "S"
$\underline{i1}(S1) = $ "TS"

$\underline{i1}(01) = $ "TS"
$\underline{i1}(02) = $ "S"
$\underline{i1}(S1) = $ "S"

Before access to 02

After access to 02

Figure 5. Low-Water Mark Policy

28

A3.3 insures that improper activation of more privileged sub_ects may not cause indirect damage to "higher" integrity level objects. The intent of A3.1 is to insure that indirect improper modifications are equally prevented. T3.1 assures us of this protection.

(D3.1) An information transfer path is a sequence of objects $<o[1],\ldots,o[n+1]>$ and a corresponding sequence of subjects $<s[1],\ldots,s[n]>$ such that

$$\forall\ i\ \in\ [1,\ldots,n]\quad s[i]\ \underline{o}\ o[i]\quad\underline{and}\quad s[i]\ \underline{m}\ o[i+1].$$

(T3.1) If there exists an information transfer path from object $o[1]\ \in\ O$ to object $o[n+1]\ \in\ O$ then enforcement of the low-water mark policy requires

$$\underline{il}(o[n+1])\ \underline{leq}\ \underline{il}(o[1]).$$

Proof: If an information transfer path exists, then D3.1 specifies that there exists the indicated sequence of subjects and objects. We assume that each access is indivisible and mutually exclusive: that is, access to a given object is deferred pending the completion of any previous access. Further, we assume each access was successful and performed in the order: observe, modify. An unsuccessful access, in the path, denies the hypothesis. For any $k\ \in\ [1,\ldots,n]$

$$\underline{il}'(s[k])\ =\ \underline{min}\ \left\{\underline{il}(o[j]\ \middle|\ 1\leq j\leq k\right\}$$

after completion of the k-th observation (A3.1). Therefore

$$\forall\ o[j]\ j\ \in\ [1,\ldots,n]\ \underline{il}'(s[n])\ \underline{leq}\ \underline{il}(o[j])$$

after completion of the n-th observation. Since the n-th modify access succeeds

$$\underline{il}(o[n+1])\ \underline{leq}\ \underline{il}'(s[n]).$$

Therefore: $\underline{il}(o[n+1])\ \underline{leq}\ \underline{il}(o[1])$.

This policy, in practice, has rather disagreeable behavior. The transition function specified by A3.1 requires the integrity level of subjects to change as a function of the objects observed. A monotonically non-increasing subject integrity level makes generalized, domain independent programming awkward at best, since the set

29

of objects modifiable by a given subject can change[10] with each
observation. In a sense, a subject can sabotage (inadvertently) its
own processing by making objects necessary for its function inac-
cessible (for modification). The problem is serious since there is
no recovery short of reinitializing the subject. However, for some
applications, this fault could be a benefit since it allows access
to a changing set of objects during a subject's existence.

A Low-Water Mark for Objects

The above discussions explicitly assume that it is the integrity
level of subjects which changes. An alternative formulation postu-
lates that it is the integrity level of modified objects which changes.
We can characterize this alternate policy by the following rules.
For each observe access by a subject s to an object o:

$$\underline{il}'(s) = \underline{min} \left| \underline{il}(s), \underline{il}(o) \right|.$$

For each modify access by a subject s to an object o:

$$\underline{il}'(o) = \underline{min} \left| \underline{il}(s), \underline{il}(o) \right|.$$

It should be apparent that the integrity level of a given ob-
ject (or subject) is monotonically non-increasing. Ill-considered
behavior on the part of subjects will result in every subject and
object possessing the lowest integrity level of any accessed object.
Further, this policy does not prevent improper modification; rather
it insures that such modifications are apparent. Thus, this policy
seems ill-suited to a computer system environment.

A Low-Water Mark Integrity Audit Policy

An unenforced variant of the above model, a "level of corruption"
model, provides a measure of possible corruption of data bases with
"lower" integrity level information. We define, for subjects and ob-
jects, a "current corruption level" (abbreviated $\underline{cl}$) which is defined
in the following manner. For each observe access by a subject s to
an object o:

$$\underline{cl}'(s) = \underline{min} \left| \underline{cl}(s), \underline{cl}(o) \right|.$$

---

[10] It can only decrease the size of the set of modifiable objects and
invocable subjects.

For each modify access by a subject s on an object o:

$$\underline{cl}'(o) = \underline{min}\left|\underline{cl}(s),\underline{cl}(o)\right|.$$

The value of $\underline{cl}$ for an object then represents the least integrity level of information which could have been used to modify the object.[11] We note that detailed analysis of the computations[11] performed by a subject can be used to refine this value.

The Ring Policy

Section II identified two classes of improper modifications: direct and indirect. The low-water mark policy dealt with the direct modification problem directly: it forbade direct modifications (A3.2 and A3.3). Indirect improper modifications were prevented by a change in the accessing subject's integrity level (and thus its accessing domain). The ring[12] policy provides kernel enforcement only of a protection policy addressing direct modification. The integrity levels of both subjects and objects are fixed during their lifetimes and only modifications of objects of less than or equal integrity level are allowed. While permitting less substantial assurances of integrity,[13] particularly concerning indirect improper modification, the flexibility of the system is substantially increased. This is accomplished by allowing observation of objects at any integrity level.

The policy is defined by two axioms.

(A3.4) $\forall o \ni O, s \in S \quad s \underline{m} o \Rightarrow \underline{il}(o) \underline{leq} \underline{il}(s)$.

(A3.5) $\forall s[1], s[2] \in S \quad s[1] \underline{i} s[2] \Rightarrow \underline{il}(s[2]) \underline{leq} \underline{il}(s[1])$.

By construction, no object of "greater" or "incomparable" level of integrity may be modified (or subject improperly invoked) by a given subject.

---

[11] That is, a trace of the sequence of data accesses actually used to modify the object.

[12] Named for its similarity to the protection mechanism provided by the Multics hardware [12].

[13] That is, without verifying properties of user programs.

31

Of course, the corruption of data bases (and programs) at a sub-
ject's integrity level is the programmed responsibility of the sub-
ject. Since no external access controls prevent possibly corrupting
observations, the subject must provide internal controls to validate
observed data. Program verification (of some form) must insure that
no such corruption occurs. The lack of constraints on observe ac-
cess does, however, allow a much wider range of discretion (and re-
sponsibility) to subjects as to the validity of observed data and
procedures.

The Strict Integrity Policy

The strict integrity policy can be considered the "complement"
or "dual" of the security policy [2] [3]. It consists of two axioms,
which, analogously to the simple security condition and *-property
[2], prevent the direct and indirect sabotage of information. In
this case as for security, this property is true only to the extent
that integrity levels are properly assigned to subjects and objects.

$\underline{i1}$(01) = "TS" $\underline{i1}$(S1) = "S"
$\underline{i1}$(02) = "C"
$\underline{i1}$(03) = "C"
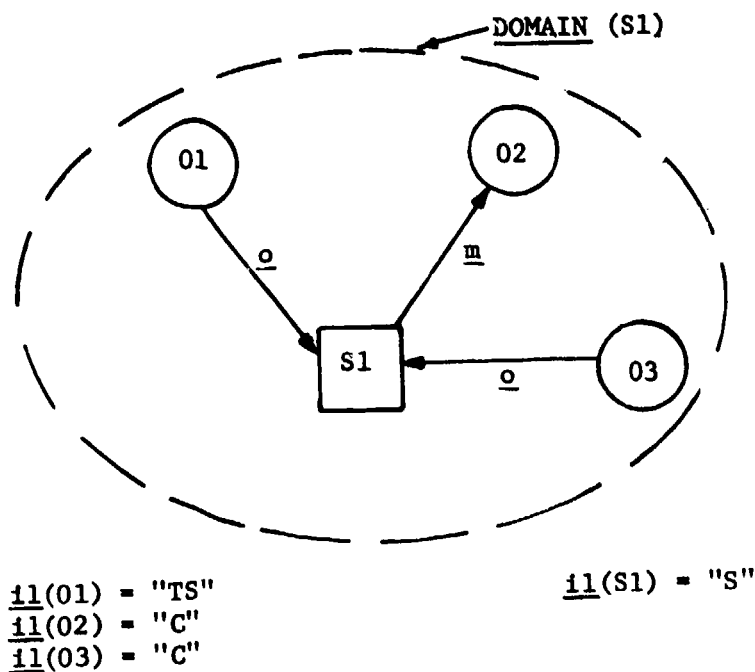
Figure 6. Ring Policy

32

In many ways, this policy provides the same capabilities as the low-water mark policy. However, where the low-water mark changes a subject's integrity level to prevent indirect sabotage, strict integrity forbids the access. This strategy permits the simple recovery from improper (observe) access: a situation not found for the low-water mark policy. However, this is achieved for the price of making many objects inaccessible (unobservable) to a given subject.

Three axioms characterize the strict integrity policy.

(A3.6) $\forall s \in S, o \in O \quad s \underline{o} o \Rightarrow \underline{i1}(s) \underline{leq} \underline{i1}(o)$.

(A3.7) $\forall s \in S, o \in O \quad s \underline{m} o \Rightarrow \underline{i1}(o) \underline{leq} \underline{i1}(s)$.

(A3.8) $\forall s[1], s[2] \in S \quad s[1] \underline{i} s[2] \Rightarrow \underline{i1}(s[2]) \underline{leq} \underline{i1}(s[1])$.

$\underline{i1}(01) = \text{"TS"}$ $\underline{i1}(S1) = \text{"S"}$
$\underline{i1}(02) = \text{"C"}$
$\underline{i1}(03) = \text{"S"}$
$\underline{i1}(04) = \text{"TS"}$

Figure 7. Strict Integrity Policy

33

The satisfaction of A3.7 insures that objects may not be directly modified by subjects possessing insufficient privilege. However, this assumes that the modifications made by an authorized subject are all at the explicit direction of a non-malicious program. Clearly, the unrestricted use of subsystems written by arbitrary users (to whose non-malicious character our user cannot attest) does not satisfy this assumption. Thus, A3.6 constrains the use of sub-systems (data or procedures) to those whose non-malicio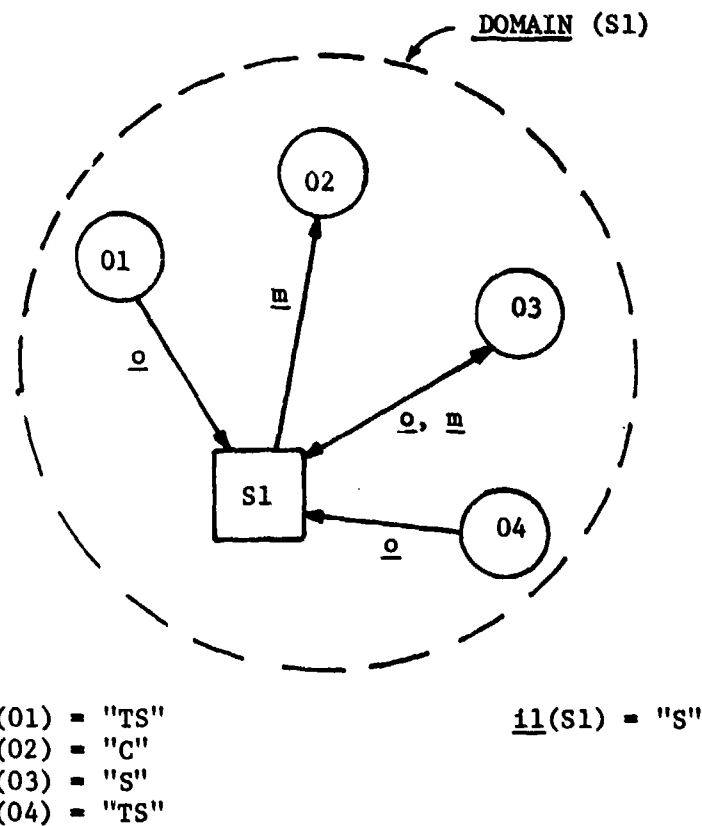us character (by virtue of their integrity level) the subject can attest: those objects having an integrity level greater than or equal to that of the subject. We can now assure ourselves that the non-malicious character of these objects is preserved by demonstrating that no information may be transferred (under the above axioms) from objects of "low" integrity level to ones of "higher" integrity level (similarly to T3.1).

(T3.2) If there exists an information transfer path from object $o[1]$ $\in$ O to object $o[n+1]$ $\in$ O then enforcement of the strict integrity policy requires

$$\underline{il}(o[n+1] \underline{leq} \underline{il}(o[1]).$$

Proof: If an information transfer path exists, then D3.1 specifies that there exists the indicated sequences of subjects and objects. From A3.6 and A3.7:

$$\forall i \in [1,\ldots,n] \quad \underline{il}(o[i+1]) \underline{leq} \underline{il}(s[i]) \underline{leq} \underline{il}(o[i]).$$

Since $\underline{leq}$ is a partial ordering, it is by definition transitive: thus

$$\forall i \in [1,\ldots,n] \quad \underline{il}(o[i+1]) \underline{leq} \underline{il}(o[i]).$$

Therefore, $\underline{il}(o[n+]]) \underline{leq} \underline{il}(o[1]).$

Thus, by virtue of T3.2, we can be assured that the strict integrity policy will maintain the integrity of objects as defined by the external assignment of integrity level.

DISCRETIONARY INTEGRITY POLICY

The preceding discussion of this section has dealt with manda-tory integrity considerations. Appropriately so, since the issue of protection of national security information is the paramount consid-eration. However, the other integrity protection problems assume

an important role for a variety of applications. Thus, this subsection will explicitly identify the mechanisms and classes of policies, other than national security integrity, to be supported by the Multics kernel.

We adopt the notation of the previous subsection to denote the model elements. Further, though the models are "dynamic" in origin, that is, the policy is a function of the system state, we present only the static constraints applicable to any one state. Dynamic considerations concerning the specification of the protection afforded data will be informally addressed.

Access Control Lists

Access control lists (ACLs) are a mechanism for the specification of the set of users whose subjects may access a given object. The contents of an access control list may be dynamically modified by an appropriately privileged subject. Hence its discretionary nature.

Model

We define the following additional model elements:

U : a set of users;

M : $\left|\text{"o"},\text{"m"}\right|$ set of abstract access modes;

<u>user</u>: S -> U, a function mapping subjects into users;

<u>acl</u>: O -> POWERSET(U x M) a function mapping objects into an element of the POWERSET of the cross product of users and access modes - the set of ACL elements;

<u>name</u>: U x M -> U, a function selecting the user component of an ACL element; and

<u>mode</u>: U x M -> M, a function selecting the access mode component of an ACL element.

The following access constraints are considered only to apply to subject -> object access. No modelling of subject -> subject invocation is intended though its derivation is not difficult. Indeed many implementations of this policy (in particular the Multics Access Control List) do not apply it uniformly to all objects.

(A3.9) ∀ s ∈ S, o ∈ O   s <u>o</u> ⊃ =>
          ∃ acle ∈ <u>acl</u>(o) <u>name</u>(acle)=<u>user</u>(s) <u>and</u> <u>mode</u>(acle)="o".

(A3.10) $\forall s \in S, o \in O \quad s \underline{m} o =$
$\exists$ acle $\in \underline{acl}(o) \quad \underline{name}(acle) = \underline{user}(s) \underline{and} \underline{mode}(acle) = "m"$.

The function $\underline{acl}$ defines the discretionary protection policy. A3.9 constrains observe accesses to those objects specifically naming the user of the accessing subject on its ACL. A3.10 similarly constrains modify accesses. The relations $\underline{o}$ and $\underline{m}$ may be changed by modifications to $\underline{acl}$. The privilege to modify $\underline{acl}$ is defined by the privilege to modify the object in which $\underline{acl}$ is located.

Protection Properties

We may now ask: What properties for integrity protection does this model provide? The problem of direct modify access is addressed by the explicit specifications of the ACL contents (A3.10). However, two problems persist: 1) indirect modification, and 2) identification of which subjects may modify the protection policy (change the ACL).

$\underline{user}(S1) = A$ $\underline{user}(S2) = B$

$\underline{acl}(O1) = \langle A, "o" \rangle$ $\underline{acl}(O3) = \langle B, "m" \rangle$

$\underline{acl}(O2) = \langle A, "m" \rangle, \langle B, "o" \rangle$
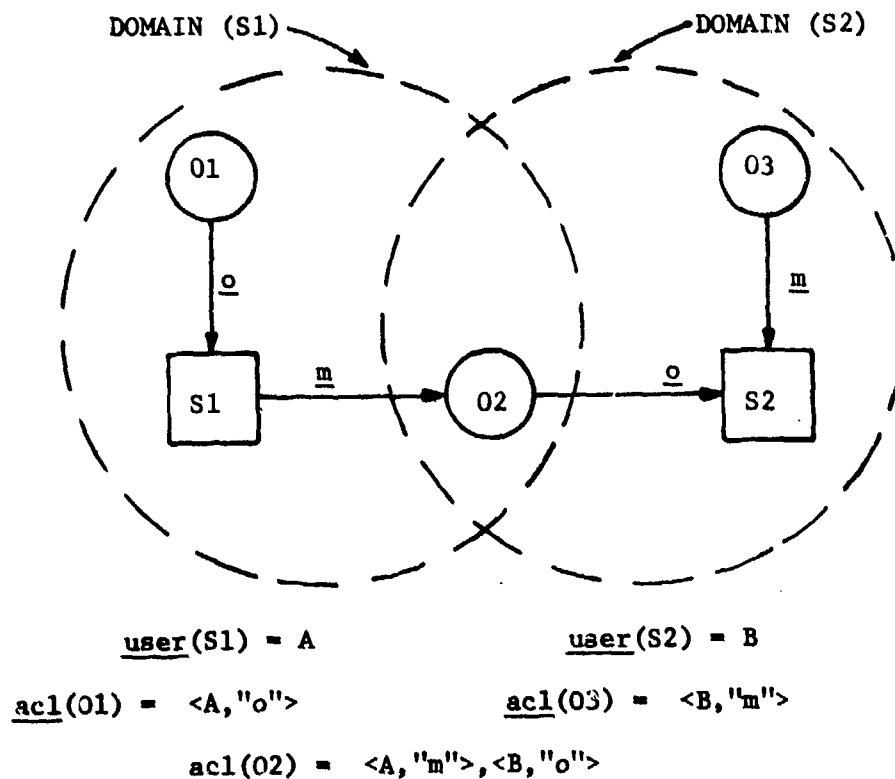
Figure 8. Access Control Lists

The problem of indirect modification is difficult to address with this policy/mechanism. The strict integrity policy made precise statements regarding indirect modification, since static, externally defined classes of subjects and objects (integrity levels) were defined. The properties externally attributed to subjects and objects of a given integrity level allowed some statement as to the preservation of these properties. However, no such statement may be easily made in this case. The problem is compounded by the dynamic nature of ACLs. Since the set of subjects (users) which may access an object is not under external control, no properties of these subjects (users) may be externally defined. Thus, no statement regarding the properties of accessed objects can be made solely on the basis of the access control mechanism. In fact, the lack of "precise statements" is a fundamental characteristic of discretionary policy, and it is this characteristic that tends to distinguish it from non-discretionary policy. The behavior of accessing subjects, particularly to which users (subjects) they delegate privilege, must also be taken into account.

Unlike integrity and security levels, the protection afforded by ACLs is dynamic. That is, the set of users and access modes, specified by an ACL (acl) may be modified by an appropriately privileged subject. For Multics, an appropriately privileged subject is one that has modify access[14] to the directory containing the branch defining a given object[15] [7]. Since the capability to change an object's ACL is represented by having modify access to its containing directory, a subject having such access may give itself direct access to the object if it does not already have such access. Indeed, we may apply this notion recursively to the file hierarchy. Thus, any subject that possesses modify access to a directory, may give itself (or other subjects) discretionary access to any element of the subtree rooted at that directory.

We may formalize this discussion by defining the directory hierarchy as done in [3] and redefining the access modes o and m in such an access hierarchy. An object hierarchy H may be defined as a rooted tree of objects: a subset of the entire object set O.

---

[14] Or that may obtain it. See discussion in the following paragraphs.

[15] acl(o) is logically contained in the directory for o. Thus, privilege to change acl is distributed throughout the system.

37

The Multics file system is an example of such a hierarchy. We may define it by the following axioms defining a relation ancestor for the tree. For o,p ∈ H ancestor(o,p) states p is an ancestor of o. This relation is quite similar to the dominates relation of [3].

(A3.11) ∀ o ∈ H ancestor(o,o).

(A3.12) ∀ o,p ∈ H ancestor(o,p) and ancestor(p,o) => o=p.

(A3.13) ∀ o,p,q ∈ H ancestor(o,p) and ancestor(p,q) => ancestor(o,q).

(A3.14) ∀ o,p,q ∈ H ancestor(o,p) and ancestor(o,q) =>
            ancestor(p,q) or ancestor(q,p).

Given the above definition, we may augment the relations o and m given that to access any object in H; a subject must have o access to all its ancestors.

(A3.15) ∀ s ∈ S, o,p ∈ H s o o and ancestor(o,p) => s o p.

(A3.16) ∀ s ∈ S, o,p ∈ H s m o and ancestor(o,p) and o≠p => s o p.

Further, we identify the set of subjects who may gain access themselves or give access to another subject, by the relation accessible(o,s) for subject o changing the domain of object o.

(A3.17) ∀ s ∈ S, o ∈ H accessible(o,s) => s o o or
            ∃ p ∈ H ancestor(o,p) and s m p.

The community of users and subjects identified by accessible are those which have or may obtain access to an object. We note that the definition of accessible and the object hierarchy can be extended to other protection policies or combinations of policies.

The above analysis states that a number of subjects, not uniquely determined, may determine the ACL contents of a given ob- ject. Thus, the modification protection afforded by ACLs in the file hierarchy is limited by the propagation of privilege. We must trust the set of subjects which have modify access to some superior directory (of a given object) not only to give themselves access (and utilize it properly) but also not to give other subjects (of other users) access. In an environment characterized by a rapid

turnover of users, and extensive data sharing, the dynamic nature of ACLs provide only limited protection against improper modification[16].

Rings

The ring protection policy/mechanism we shall discuss is the prototype for the ring policy discussed in the previous subsection. It offers the capability to

a) protect a subsystem from its invoker; _or_

b) protect an invoker from an invoked subsystem.

It cannot provide both capabilities for any combination of invoking and invoked subsystem.

The protection schema we describe differs from the mandatory ring policy in the following respects:

1) it is hardware supported by the Multics processor;

2) it is applied to different system elements; particular subjects are mapped to intraprocess procedures rather than processes;

3) provision is made for the modification of access privilege (changing of ring allocation); and

4) greater flexibility for intersubject invocation is provided.

Model

We define the following elements:

R: a finite set of ring names, normally small integers;

---

[16]A useful utility, for a given implementation, would be a function to identify the current set of users which might potentially gain access to a given object. From our definition of the relation _accessible_, this function would have the value

$$\Big| \text{user(s)} \ \Big| \ s \ \in \ S \ \text{and accessible(o,s)} \Big|.$$

Of course, this information may only be given to subjects which themselves have access to the object.

≤:   a linear ordering, "less than or equal", on R;

<:   a linear ordering, "less than", on R;

r: a function, S -> R, defining the ring of execution for each subject;

lir: a function, S -> R, defining the lower ring bound for each subject's invoker;

uir: a function, S -> R, defining the upper ring bound for each subject's invoker;

umr: a function, O -> R, defining the upper ring bound for modify access; and

uor: a function, O -> R, defining the upper ring bound for observe access.

DOMAIN (S2)   DOMAIN (S1)

$\underline{r}$(S2) = 1 $\underline{r}$(S3) = 0 $\underline{r}$(S1) = 4

$\underline{uor}$(O1) = 4 $\underline{uor}$(O2) = 2 $\underline{uor}$(O3) = 0
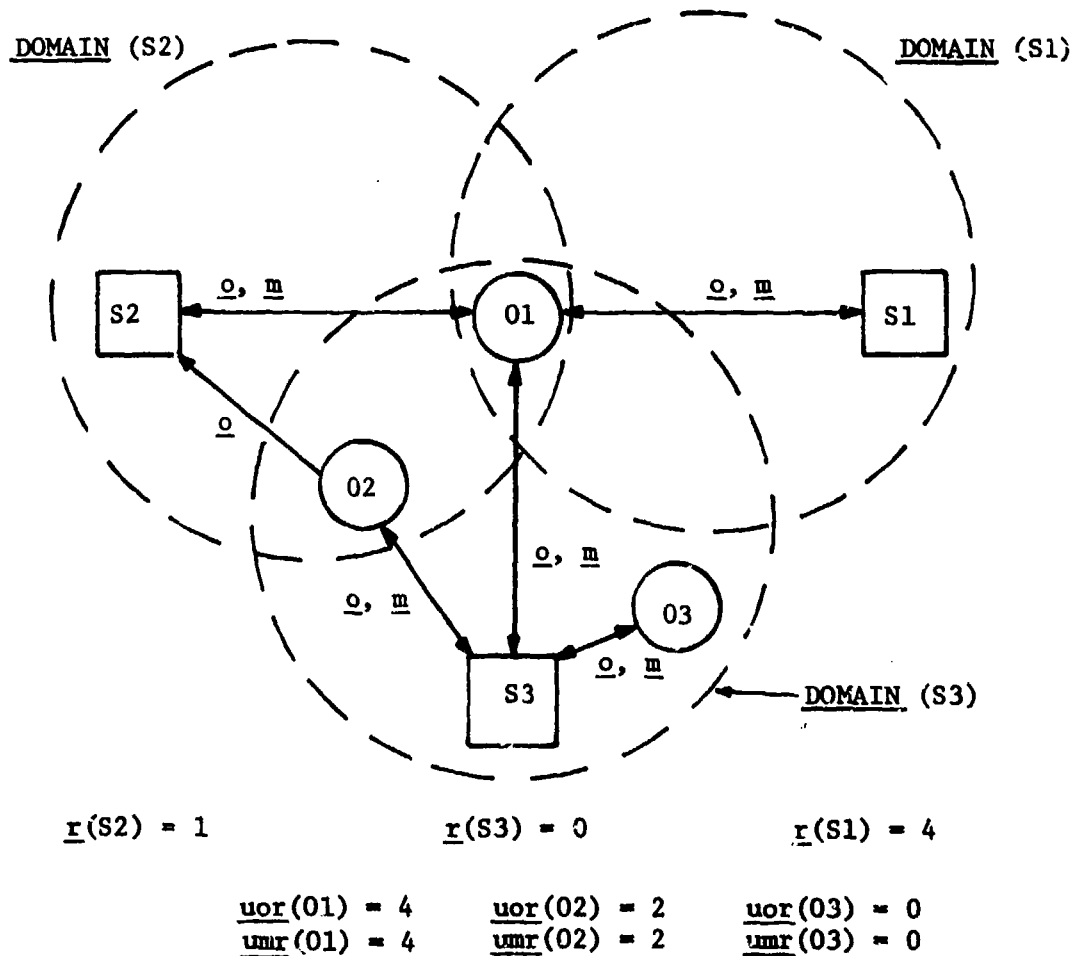$\underline{umr}$(O1) = 4 $\underline{umr}$(O2) = 2 $\underline{umr}$(O3) = 0

Figure 9.   Rings

40

The constraints on access are formalized by the following axioms:

(A3.18) $\forall s[1], s[2] \in S \quad s[1] \underline{i} s[2] \Rightarrow$
$\quad (\underline{lir}(s[2]) \leq \underline{r}(s[1]) \leq \underline{uir}(s[2]) \text{ and }$
$\quad \underline{r}(s[2] < \underline{r}(s[1])) \text{ or }$
$\quad \underline{r}(s[1]) \leq \underline{r}(s[2])).$

A3.18 constrains a subject to invoke subjects:

1) of greater privilege only through an allowed range of rings; and

2) of less than or equal privilege indiscriminately.

(A3.19) $\forall s \in S, o \in O \ s \underline{o} O \Rightarrow \underline{r}(s) \leq uor(o).$

A3.19 constrains subjects to only observe objects in an allowed range of rings: $C \leq \underline{r}(s) \leq \underline{uor}(o).$

(A3.20) $\forall s \in s, o \in O \ s \underline{m} o \Rightarrow r(s) \leq \underline{umr}(o).$

A3.20 constrains subjects to only modify subjects in an allowed range of rings: $0 \leq \underline{r}(s) \leq \underline{umr}(o).$

We consider each subject to have a single point of invocation (entry),[17] A3.18 constrains intersubject invocation to a specific "band" of domains (rings). While A3.18 assumes an explicitly specified lower bound, A3.19 and A3.20 implicitly assume a lower bound of the least element of R.

We note, as for access control lists, that this mechanism may not apply uniformly to all subjects, objects, and modes of access within a specific implementation.

Protection Properties

The analysis provided above for access control lists applies to the ring protection structure. Like ACLs, the protection policy may be altered by an appropriately privileged subject. The ability to alter the protection attributes of a subject or object must be

---

[17]This structure may be mapped onto the Multics "gate" structure (where multiple entry points are defined per gate) by considering each entry point to define a unique subject. In this view, multiple entry points act only as mechanism to conserve space.

restricted to subjects of greater than or equal privilege. The
manner in which access is restricted is dependent upon the manner
in which subjects and objects are organized. Indeed, for Multics,
we find a similar formulation of possible access (as for ACLs) may
be made, particularly within an object hierarchy. Thus, its
discretionary nature poses problems of identifying who does what to
whom.

However, the finite (eight for Multics) size of the set of
possible domains provides room for hope. Careful control (and
limitation) of the set of privileged subjects (particularly in the
"low" rings) allows knowledge of protection behavior. Further, the
set of subjects and objects (in most applications, particularly in
"low rings") remains static. The properties of static subject and
object population (which allows proof of behavior to be applied to
these) and limited number of linearly ordered domains permit useful
application of the ring structure for integrity protection.

SECTION IV

APPLICATION

The integrity policies described in the previous section may now be applied to a prototype computer system. The system of particular concern is a secure, kernel-based Multics. Our investigation will consider the two integrity environments identified in Section II: 1) the kernel and 2) the kernel-defined virtual (user) environment. For each, we will indicate applicable integrity policies for identified integrity threats.

We begin this section with a brief overview of the mapping of model elements to the Multics environment. Of particular concern is an identification of the protection mechanisms available within Multics, and which policies are enforceable by which mechanisms. We then consider integrity issues within the kernel and conclude with integrity issues within the virtual environment.

MULTICS ACCESS CONTROL STRUCTURE

Protection Mechanisms

The current Multics hardware base[18] supports two classes of protection mechanisms: 1) descriptor segments and 2) rings of protection. The first defines virtual name spaces which provide (through a binary, yes/no, decision) mappings from virtual names to physical objects resident on some system storage. The second partitions each of these name spaces into eight equivalence classes and provides the discretionary ring access controls, discussed in the previous section, between these sub-name spaces. The protection characteristics of any name within a name space are determined by the value of a descriptor for that name within the descriptor segment. In particular, the descriptor provides two facilities: 1) an access capability (for all subjects using this name space) defining all allowed modes of access to this name, and 2) constraints on that access based on the ring attributes of the accessed name (defined in the descriptor) and of the accessing subject.

There are two mechanisms for intersubject invocation. The first is the procedure call: a subject, executing within a ring within a given name space, invokes another name, as a subject,

---

[18] The Honeywell Information Systems' 6100 series and newer level 68.

43

within the same name space. The invocation may (or may not) be of
a subject within a new ring; however, the name space remains the
same. The second mechanism allows transfer from one name space to
another through the change of a Multics processor name space
pointer. For the current implementation of Multics, and for the
proposed kernel-based system under design, structural considerations
(and hardware support) make the former of these mechanisms the most
efficient. We find the procedure call within a name space (con-
strained by the ring protection mechanism) to provide us with the
kernel invocation mechanism.

Subject Structure

A central concern in the mapping of model elements (particular-
ly subjects) to an implementation is a match between the capabilities
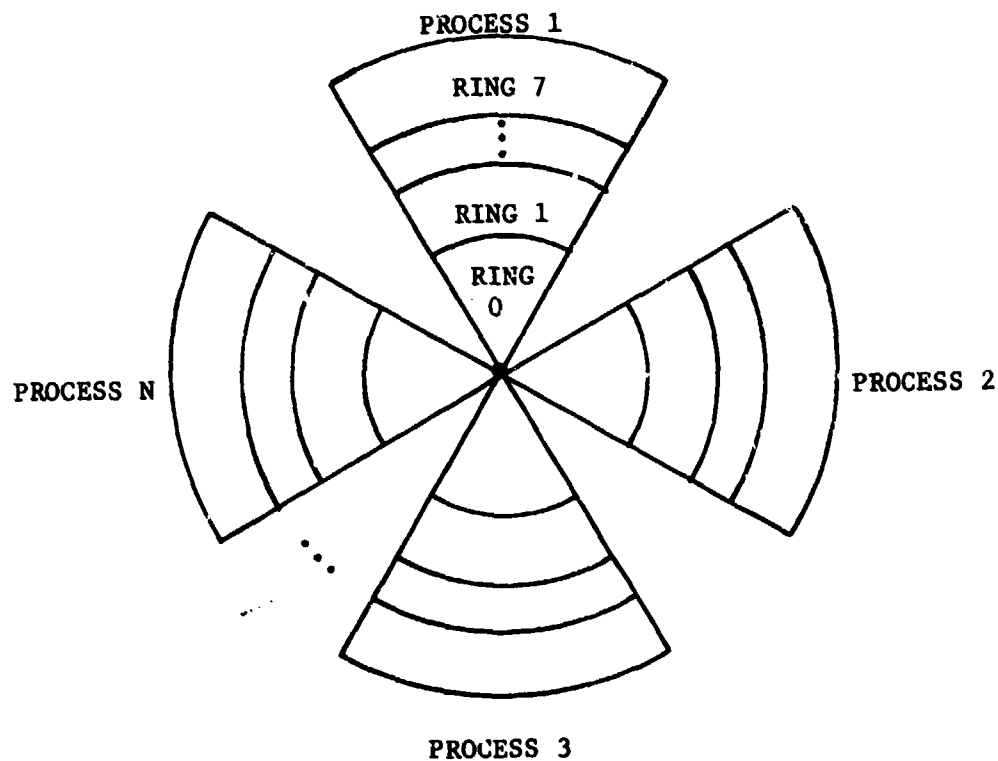of the provided protection mechanisms with both the requirements of

Figure 10. Subject/Process Structure

44

the provided protection mechanisms and the requirements of each subject's protection policy. The discussion of Section II indicated three protection problems (and policies) of interest: 1) mandatory, national security information integrity; 2) discretionary integrity based on user identity; and 3) subsystem integrity, particularly of the kernel.

The first two policies, in use, are characterized by a relatively low rate of domain change (if any). The granularity of access, for these subjects, applies to all uncertified, user-supplied subjects. In addition, a process is associated with a user. Thus, the protection properties of a process should be derived from those of its user. The properties of the name space mechanism, above, seem to match quite closely to these properties. Likewise, the protection requirements of specific subsystems within these subjects seem to be best addressed by the intraname space ring protection mechanism.

A Multics process is then a collection of subsystems, each of which is a subject constrained by the discretionary ring mechanism, the union of which can be considered a subject constrained by a name space (domain) whose mapping to physical objects is determined by the mandatory protection policies (security and integrity) and the discretionary user identity (ACL) policy. The following two subsections will describe in greater detail the protection policies applicable for both these subject types. Of particular concern for subs ems is the set of subjects and objects which compose the protection kernel. From the above discussion, the kernel is composed of subjects distributed among all Multics processes (name spaces = descriptor segments).

KERNEL INTEGRITY

The notion of kernel integrity encompasses the "tamperproof" isolation property formulated as a reference monitor requirement. Tamperproof implies that the kernel domain is protected from modification which ould result in a functionality differing from that derined by the kernel specification. Beginning with the assumption of initial correct kernel operation, a kernel integrity policy must specify the criteria which, if followed, will preserve the correctness (integrity) of the kernel. Section II defines the kernel as composed of s cts and objects resident in the most protected ring domains ach process (name space). Our attention is therefore focused on protection within a process name space, provided by the ring structure.

45

We begin our analysis with an enumeration of the methods of improper kernel modification (integrity threats) and conclude with the required modification policy.

Kernel Threats

The integrity threat categories suggested in Section II will be used to organize our analysis. We consider both external and internal kernel threats, with direct and indirect subspecies.

External Threats

External threats primarily arise from incomplete or erroneous kernel perimeter specifications, particularly with respect to implicit sharing of data or procedure between the kernel and its invokers. Direct threats may be summarized by two general attacks: 1) direct modification and 2) kernel entry at other than intended entry points.[19] Indirect threats may be similarly listed as: 1) incomplete kernel argument validation, and 2) implicit/explicit use of non-kernel data or procedure.

Direct modification is the direct access (via a read or write operation) to a kernel data base (or procedure) and the subsequent performance of improper modifications (viz., any modification which alters the non-kernel visible behavior of the kernel).[20] Kernel entry at other than the intended gates[21] allows the kernel invoker the capability to alter the kernel function from that intended. Such entry, in essence, defines new kernel functions performing un-intended functions with the access privilege of the kernel subsystem.

Arguments to kernel functions (subsystem entry points) must satisfy two criteria: 1) the arguments must be located in objects accessible to the function's invoker, and 2) the value of the argument must correspond to that specified for this function (and which the kernel's correctness proof assumes). Satisfaction of these criteria insure that the kernel cannot be induced to misuse its privilege (by accessing an object its invoker may not access) and

---

[19]Invocation of uncertified subjects, and hence of unknown (possibly malicious) behavior, with kernel access privileges.

[20]That is, the behavior specified by the top level formal specification of the kernel's function [13].

[21]Gate is the Multics term for a domain entry point.

its function cannot be changed by improper arguments (we assume the completeness of the formal verification with respect to the values of the function's arguments). An extension of this concern is the kernel use of non-kernel data or procedure. A particular example is the use of system-wide library routines (viz., a square-root routine) which may be modified by non-kernel software.

Internal Threats

Assuming the initial correctness (consistency with the specification) of the kernel, internal threats arise from two sources: 1) false assumptions and 2) improper mapping of kernel names. The primary source of false assumptions relates to hardware behavior. The a priori specification of perhaps uncertifiably reliable hardware implies that assumptions about its behavior must be made. These hardware assumptions (as is the wont of hardware) may be invalidated by quite transient conditions. The improper mapping of internal kernel names to hardware elements is the other source of internal threats. We assume that each kernel data object (uniquely named) has a unique physical representation. If this is not so (viz., a working data area mapped onto kernel procedure code) the kernel state may no longer satisfy the verification conditions initially proved.

Kernel Policy

External Threats

The direct threats are, appropriately, the most directly addressable. The ring protection mechanism/policy supplied by the Multics hardware is designed specifically for this purpose. First, we require that the data bases and procedures constituting the kernel be assigned a write-ring such that modification may not be performed outside the kernel ring(s) (domain: probably composed of rings 0 and 1). The second direct threat can be addressed by: 1) assignment of "execute" brackets (for kernel procedures) totally within the kernel domain, and 2) careful restriction of kernel gates (invocable subjects) to only those operations defined by the kernel specification. Thus, a ring-triple of <1,1,3>[22] for kernel gates, and <0,1,1>) for kernel data bases and internal procedures would be appropriate. Of course, the set of kernel segments must be explicitly defined, changing (in a precisely defined manner) only to construct those kernel data bases which implement kernel defined objects (particularly process definitions).

---

[22] We assume a two-ring kernel, rings 0 and 1, and a two-ring perprocess operating system, rings 2 and 3. See reference [12] for ring technology.

Arguments to kernel operations (both input and output) must be minimally validated as derived only from data accessible (respectively observable and modifiable) in the (non-kernel) domain of the invoking user process. The Multics hardware, through use of the ring structure to isolate kernel and non-kernel domains, provides a convenient mechanism for this validation. Also, the proper value for these arguments must be validated at each kernel access to them. This may be most directly accomplished by copying, at function entry, into kernel domain objects and validated once at entry. Thus, the arguments are protected from modification by non-kernel subjects. However, the switching of descriptor segments (name spaces) and the use of kernel data structures global to all processes must be shown not to compromise integrity, particularly between distinct non-kernel domains.

The indirect threat posed by kernel use of non-kernel data and procedure may be most directly approached by avoidance: the kernel (other than function arguments discussed above) does not use such data or procedure. The practicality of this suggestion may only be tested in a concrete implementation. However, at this time it would appear that the only problem would arise with library procedure used by the compiler in which both the kernel and the non-kernel software are primarily written. Since these procedures are not modifiable outside the kernel, a single kernel copy is all that is needed.

Internal Threats

The possibility of hardware malfunction, while not discounted, cannot be addressed as directly as software integrity. Clearly, the hardware may itself be considered as a distinct "level" of reference monitor (implementing a software defined policy contained in the descriptor segments). Thus, all of the above considerations apply.[23] However, the issue of hardware "correctness" is not easily addressed. Unlike the kernel software, the hardware has not been designed with proof in mind (viz., well-structured to avoid combinatorial explosion). The feasibility of hardware (or ill-structured firmware) validation has not been demonstrated. Exhaustive testing, in the

---

[23] For example, the issue of direct modification of hardware may seem absurd; however, the advent of dynamically microprogrammed architectures makes the issue of "firmware" modification substantial.

absence of well-structured hardware, may be the only feasible methodology for the validation of hardware (firmware) properties.[24] Further discussion of hardware reliability is beyond the scope of this investigation.

The prevention of kernel name mis-mapping may only be prevented by careful enumeration of all kernel names and the appropriate modes of access to them. The actual implementation must then be rigorously verified to maintain the relationships (and protection) defined by this enumeration. The provision of name space "firewalls", of protection domains within the kernel, could aid in this endeavor. Firewalls may be constructed (for Multics) by partitioning the kernel into multiple rings (thus protecting the inner rings from all outer rings) and by the provision for multiple name spaces within the kernel. Multiple name spaces may be constructed by providing multiple, asynchronous processes (within the kernel) to define kernel facilities. Examples within the current Multics design include the resource schedulers.

VIRTUAL ENVIRONMENT INTEGRITY

The virtual environment, as previously defined, is composed of those subjects that do not constitute the protection kernel. The access domains of these subjects are constrained by several integrity policies: 1) a mandatory policy addressing the integrity of national security information; 2) a discretionary policy addressing user identity access specification; and 3) subsystem protection. The policy provided for subsystem protection is that used for kernel integrity maintenance. Similar considerations apply to any user-defined subsystem. The properties of access control lists were extensively investigated in the previous section, and in the literature on Multics. We shall not address these policies further in this paper.

Thus, in this subsection we will be particularly concerned with the identification of a suitable mandatory integrity policy and its application to the Multics virtual environment.

---

[24] We assume no eagerness to redesign (and reimplement) the hardware in a well-structured manner.

A Recommended Policy

The mandatory models defined in Section III do not exhaustively
enumerate the space of models. However, the set does represent a
preliminary culling of worthwhile protection notions in the DoD
environment. With that caveat, some summary analysis and selection
of the "best-suited" policy is in order.

None of these models addresses all of the integrity protection
problems in the virtual environment. The discretionary models
address another portion of possible problems. However, many pro-
blems are highly application-specific, thus beyond the scope of
this investigation. A mandatory model is addressed to one purpose:
protection from malicious modification. It has little effect with
regard to prevention of accidental improper modifications.[25] Our
goal may only be the construction of a policy which enforces integ-
rity protection commensurate with the appropriateness of external
assignment of integrity levels to users, data, and programs. In
this light, we suggest the strict integrity model as most suitable.

The policy is selected for the following reasons. First, it
provides the greatest practicable protection. The constraint with
respect to "reading down" (A3.6) provides a significant barrier
against the placement of Trojan Horses (or other forms of malicious
modification). While similar protection is provided by the low-
water mark policy, the behavior of that policy, in regard to a
changing subject domain (integrity level), makes its practical use
undesirable. Second, the protection provided is relatively easy to
understand. Recent results [14] indicate the difficulty in formu-
lating effective arbitrary access control policies. A policy, such
as strict integrity, that is simple and universally applied is the
easiest to certifiably enforce.

However, the utility of strict integrity is dependent on its
use in a "real" environment. While T3.2 does insure some protection,
it is commensurate with the properties, externally enforced and
assigned, attributable to integrity levels. As indicated above, it

---

[25]The anecdote describing the production of Hamlet by a particularly
long-lived simian is appropriate here. An accidental error, in a
suitably privileged program, will sometimes cause a modification
labelled improper. As indicated earlier only fine-grained (appli-
cation dependent) access control provided by system external certi-
fication can be of assistance. Such certification can be provided
by program verification techniques.

50

cannot address (except peripherally) accidental modification. Thus, the utility of the policy must be judged by its use. In an environment characterized by the presence of a number of subsystems (composed of both data and procedure) that require unique levels of modification protection and in which there is much cross-level sharing of data and procedure objects,[26] the strict integrity policy finds its greatest utility. However, some environments may find its constraints too cumbersome for practical use.

A viable alternative for such environments is the use of the ring policy. The effective removal of A3.6 allows greater programming freedom while retaining protection against direct sabotage. However, the responsibility for protection against indirect sabotage (viz., placement of Trojan Horses) must rest with programming convention or explicit user execution time checks (viz., validation of the proper library routine). The policy provides no explicit protection against indirect integrity threats. The impact on integrity control caused by the enforcement of the ring policy rather than strict integrity is related to the impact on security caused by the removal of *-property [2] enforcement. In each case, indirect protection threats must be addressed by access controls outside of the kernel. Since no assurances as to the proper behavior of non-kernel access controls can be made, no effective protection against indirect threats can be provided.

The ring policy may even prove too restrictive or cumbersome for some environments. For such instances, the integrity policy may be incorporated into the security policy. One [2] formulation of the security policy requires that a subject may modify objects only of equal security level. Thus, the security level becomes a form of integrity policy, preventing direct modification from subjects of "lesser" security level. However, as will be shown below, this policy cannot address certain problems posed by hierarchical file systems, or the differing protection requirements for shared data bases, discussed earlier.

---

[26]The benefit is maximized when sharing is distributed over a number of integrity levels. If the shared objects are all "global" objects, which must then by convention be placed at a "system high" integrity, then the convention has, in practice, usurped the function of A3.6. With such global sharing, other mechanisms (notably ACLs) may be used to effect proper protection.

The enforcement of any of the above policies might prove impractical in some benign environment. For such systems, the low-water mark integrity audit policy provides a tool for the administrative control of integrity corruption. Its use as an audit mechanism may prove useful to a community of systems supporting information sharing in a generally benign environment. However, it must be noted that auditing provides no protection, only _post facto_ notification of an integrity compromise. Thus, it is useless for the _prevention_ of integrity compromise.

Virtual Environment Impact

Our preliminary analysis of the impact of the imposition of the strict integrity policy focuses on two issues:

1) application of the strict integrity policy to the classes of objects supported by the Multics kernel, and

2) some indication of the impact on user behavior (particularly the assignment of integrity levels).

We discuss each in turn.

Application

We briefly consider the application of the strict integrity policy to the functional Multics environment provided by the security kernel. Our analysis is confined to the primary kernel subsystems of process control/communication, on-line storage control and external I/O. For each functional area, the specific application of A3.6 — A3.8 is presented.

Let us consider the set of subjects that compose the user environment of a Multics process. Each process is assigned an integrity level equal to the integrity level of its component subjects. The process control functions of _create_, _delete_, and _wakeup_ are all instances of the model operation of invocation, operating on other processes. Hence, A3.8 constrains the access domain. Thus, the integrity constraints on $\underline{i}$, for any process p invoking a process p', are realized as:

$$\underline{i}(p') \ \underline{leq} \ \underline{il}(p).$$

52

We identify an external I/O socket as an instance of an object. An I/O socket is a logical I/O stream that acts as a data and control path between a Multics process and an I/O device. Each socket, for each process, is assigned an integrity level. A process p is then constrained to <u>send</u> (an instance of modification satisfying <u>m</u>) only over a socket sk where:

$$\underline{il}(sk) \ \underline{leq} \ \underline{il}(p).$$

A process p' may <u>receive</u> only over a socket sk' where:

$$\underline{il}(p') \ \underline{leq} \ \underline{il}(sk').$$

We identify a segment as an instance of an object. Each segment is assigned an integrity level.[27] A process p is then constrained to <u>read</u> or <u>execute</u> (including <u>call</u> and <u>return</u>) on those segments s where:

$$\underline{il}(p) \ \underline{leq} \ \underline{il}(s).$$

A process p' may only write or append those segments s' where:

$$\underline{il}(o') \ \underline{leq} \ \underline{il}(p').$$

We identify directories as instances of objects. A process p may only have <u>search</u> or <u>status</u> access to those directories d for which:

$$\underline{il}(p) \ \underline{leq} \ \underline{il}(d).$$

Message segments constitute not only a multi-security level object, but also a multi-integrity level object. Individual messages, as well as message segments, possess an integrity level. A process p may <u>enter</u> a message m into a message segment ms (in directory d) where:

$$\underline{il}(m) \ \underline{leq} \ \underline{il}(p) \ \& \ \underline{il}(ms) \ \underline{leq} \ \underline{il}(m) \ \& \ \underline{il}(m) \ \underline{leq} \ \underline{il}(d).$$

---

[27] We note that in a manner analogous to other attributes, the segment's integrity level implicitly possesses (from the directory structure) integrity and security level attributes. We will discuss possible assignment of such attributes below.

A process p may remove a message m from message segment ms (in directory d) where:

$il(p)$ leq $il(m)$.

The constraints of the strict integrity policy dictate the placement of integrity level constraints on the file hierarchy, similar to those introduced for security [3]. We require that for any directory d, its branches b (indicating subordinate objects: directories, segments, and message segments) must have an integrity level where:

$il(b)$ leq $il(d)$.

We can arrive at this constraint from some rather basic integrity considerations. Within the directory-based environment, a file's name may be considered as a vector composed of all branch names of its superior directories (plus its own branch name). Clearly, changing any component branch name to indicate a different subtree is a possible act of sabotage which we wish to avoid. Thus, the entire filename (including the names of its superior directories) must have at least the integrity level of the concerned object. Considered piecewise, the name of each object's superior directory (which contains the object's name) must have at least the integrity level of the object. Applied recursively from the root to the leaves, we obtain the result: the integrity level of objects, in the file hierarchy, is monotonically non-increasing, considered from the root to the leaves.

Impact

The impact of the strict integrity policy on user programming behavior is briefly considered below. We are concerned with the assignment of integrity levels within the file hierarchy, particularly in relation to the assignment of security levels.

We begin by examining useful combinations of security and integrity constraints. Figure 11 illustrates allowable modes of access for different combinations of constraints. Not surprisingly, we note that the greatest privileges accrue when both subject's and object's integrity and security levels are respectively equal. Further, the most useful access (observe) holds only if:

$sl(o)$ $\leq$ $sl(s)$ & $il(o)$ leq $il(s)$.

	$\underline{sl}(s)<\underline{sl}(o)$	$\underline{sl}(s)=\underline{sl}(o)$	$\underline{sl}(o)<\underline{sl}(s)$
$\underline{il}(s) \ \underline{leq} \ \underline{il}(o)$		$\underline{o}$	$\underline{o}$
$\underline{il}(s) \ \underline{eq} \ \underline{il}(o)$	$\underline{m}$	$\underline{o},\underline{m}$	$\underline{o}$
$\underline{il}(o) \ \underline{leq} \ \underline{il}(s)$	$\underline{m}$	$\underline{m}$	

Figure 11. Security and Integrity Constraints

The application of the strict integrity policy, in conjunction with the security policy implies, from previous discussion, two properties of the file hierarchy: 1) monotonically non-decreasing security level, and 2) monotonically non-increasing integrity level, considered from the root to the leaves. Thus, if both attributes vary from the root to the leaves in the indicated manner, for a given process p (with fixed $\underline{sl}(p)$ and $\underline{il}(p)$) a subtree may become inaccessible to some subjects because: 1) the security level is too "great," 2) the integrity level too "small," or 3) some combination.

This property is illustrated[28] in Figure 12. Process p cannot observe A.A since A.A's integrity level is too low, cannot observe A.B since A.B's security level is too high, and cannot access A.C at all for both reasons.

A typical object hierarchy using integrity levels to protect a number of distinct applications having differing levels of integrity

---

[28] The security levels of Figure 12 are labelled as follows:
U=UNCLASSIFIED, S=SECRET, and TS=TOP SECRET. These are ordered by $\underline{sl}$: $\underline{sl}(U)<\underline{sl}(S)<\underline{sl}(TS)$.

$\underline{i1}(p)$ = "S"
$\underline{s1}(p)$ = "S"

$\underline{i1}(A)$ = "TS" $\underline{s1}(A)$ = "U"
$\underline{i1}(A.A)$ = "C" $\underline{s1}(A.A)$ = "U"
$\underline{i1}(A.B)$ = "TS" $\underline{s1}(A.B)$ = "TS"
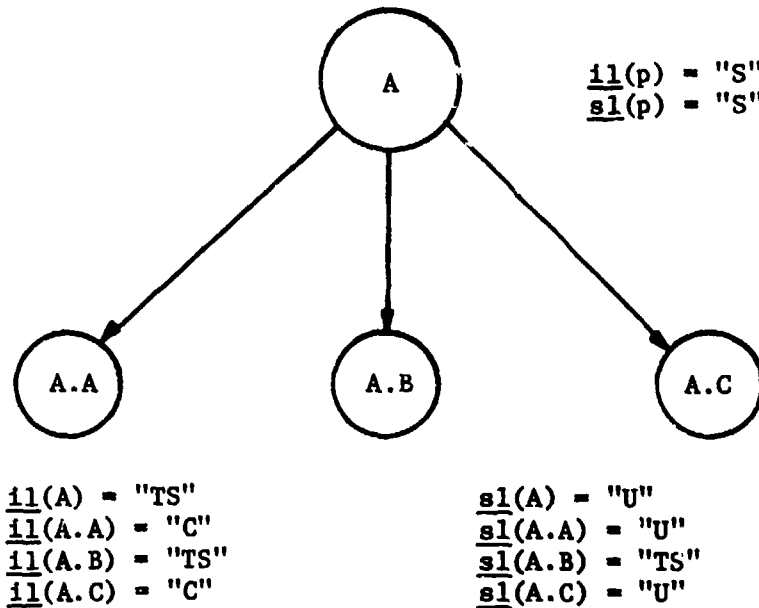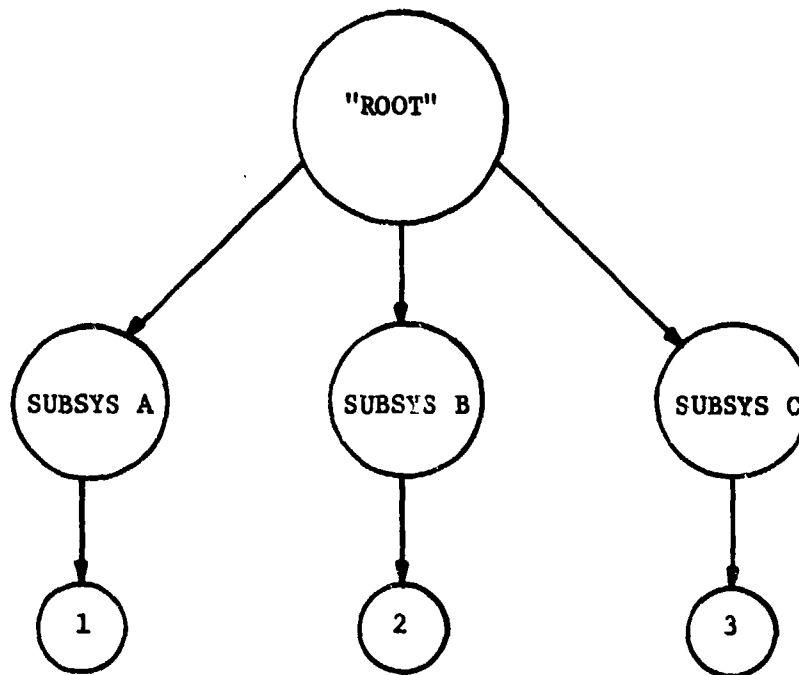$\underline{i1}(A.C)$ = "C" $\underline{s1}(A.C)$ = "U"

Figure 12.   Inaccessible Objects

is illustrated in Figure 13. The TOP SECRET root directory contains
three subsystems each at a distinct integrity level. The directory
"subsys_A" contains the segments for a CONFIDENTIAL level application.
The directory "subsys_B" contains the segments for a SECRET level
application. The directory "subsys_C" contains the segments for a
TOP SECRET level application. By construction, the CONFIDENTIAL and
SECRET applications cannot interfere with the operation of the TOP
SECRET application: its objects (segments and directories) are
inaccessible.

Verification Considerations

The verification of kernel security properties has two compo-
nents. First, we must verify that the kernel implementation satisfies
the properties defined in the formal specification. Second, the ab-
stract objects accessed by kernel primitives (viz., object attributes
including security and integrity levels) must be accessed in a mode
commensurate with their implicit or explicit security and integrity

56

$ll("root") = "TS"$ $ll("root.subsys\_A.1") = "C"$
$ll("root.subsys\_A") = "C"$ $ll("root.subsys\_B.2") = "S"$
$ll("root.subsys\_B") = "S"$ $ll("root.subsys\_B.3") = "TS"$
$ll("root.subsys\_C") = "TS"$

Figure 13. Typical Object Hierarchy

levels.[29] The verification technique developed for the security
policy is described in [15]. We shall briefly describe its extension
to the strict integrity policy.

The first problem poses no conceptual difficulty. The only addi-
tion the integrity policy makes are the constraints specified by A3.6
– A3.8 for each access. For each kernel operation, specific

---

[29] The implicit protection attributes (including those required for
all kernel-supported protection policies) are determined by the set
of objects accessed (and the mode of access) by each kernel opera-
tion.

assertions incorporating one or more of A3.6 - A3.8 must be included in the formal specification. Similarly to security assertions, the implementation must be verified to satisfy these constraints.

The second problem is also addressed by an extension of the current methodology. Each kernel-defined object observable at the kernel perimeter must be shown not to improperly transmit or modify information through kernel operations. The established methodology, for security, proceeds by the derivation of security levels for these kernel-defined objects and shows that their manipulation by kernel primitives does not cause information compromise (viz., satisfies the constraints of the security policy). Our concern is to show that their manipulation does not also cause information sabotage.

The established verification methodology [15] may be applied to any protection policy based on a partially ordered set of protection levels that defines similar access relations. Since both the set of security levels and the set of integrity levels are partially ordered, their product $(A = SLxI)$, a set of access levels, is also partially ordered. We can define a partial ordering le on A where for any two access levels al=<sl,il> and al'=<sl',il'>:

al le al' <=> sl $\leq$ sl' and il' leq il.

The access relations may be redefined as:

s o o => al(o) le al(s) => sl(o) $\leq$ sl(s)

s m o => al(ε) le al(o) => sl(s) $\leq$ sl(o)

s[1] i s[2] => al(s[1]) le al(s[2]) => sl(ε[1]) $\leq$ sl(s[2]).

These definitions encompass both the strict security and strict integrity policies and are precisely analogous to the access relations for the security policy alone. Since

1) the set of access levels is partially ordered, and

2) the access relations using access levels are precisely analogous to the security policy,

the verification methodology used for the security policy can be used for both the security and strict integrity policies.

SECTION V

CONCLUSION

The preservation of the validity of information stored in resource-sharing computer systems is a major system design question. The issues of which information to protect, from whom, and with what mechanism are complex questions requiring careful analysis. This paper has addressed these questions for a number of important protection problems found in computer utilities. The integrity protection issues for a security kernel-based Multics provided the specific context.

We began our investigation with an examination of the abstract notion of integrity. The concept of integrity as the maintenance of information validity was discussed and accepted. Specific integrity maintenance problems for a computer utility (particularly a kernel-based Multics) were identified and access control policies and mechanisms addressing solutions to these problems were formally specified. Finally, the application of these policies within the Multics environment was described.

Three protection problems were explicitly addressed:

1) the integrity protection of information vital to national security;

2) the integrity protection of information vital to the needs of a particular user; and

3) the integrity protection of the security kernel.

Access control policies providing effective protection were identified for each. These policies may be usefully applied to protection problems other than those addressed here. Appendix I identifies one such case.

The imposition of any information protection policy often has some impact on the user programming environment. Protection policies, by intention, restrict the behavior of programs. The apparent programming difficulty arising from these restrictions must be carefully evaluated against the considerable value of effective access control. The policies discussed in this paper provide a substantial tool to system administrators, designers and programmers for the proper confinement of programs to the least privilege required for their successful legitimate operation. Programming difficulty is often transient and disappears once protection policies become

familiar to users. However, protection problems are not so kind.
Ignored protection problems become security and integrity breaches.
The only way to prevent information compromise is to impose effec-
tively enforced security policies. The only way to prevent information
tion sabotage is to impose effectively enforced integrity policies.

We have directed our attention in this paper to a limited number
of integrity problems. No claim for completeness is made. However,
the developed integrity policies may usefully be applied to a much
larger class of problems.

APPENDIX I

THE CAPABILITY POLICY

The requirement that a security kernel support trusted processes
[2] or complex facilities poses a protection problem not explicitly
addressed by the policies discussed heretofore. These entities are
processes (collections of subjects and objects) to which the kernel
provides special privileges. These privileges are required so that
these entities might define extended secure facilities not ordinarily
provided by the security kernel proper. Examples of these facilities
include the system root process (Answering Service [7]), spooling
manager (I/O Coordinator), and system security officer/administrator.
Each of these facilities require special privileges, most often viola-
tion of the security *-property for one or more kernel operations, in
order to properly perform their function. Our particular protection
problem is to insure that each of these facilities is extended the
least privilege necessary to perform its function. The special
functions and privileges required are provided by special kernel
operations. We need to define an integrity policy that restricts
these special kernel operations only to those facilities that require
them. The capability policy provides this service.

We define the following policy elements:

S: the set of subjects;

O: the set of objects;

C: the set of capabilities;

caps: S -> POWERSET(C), a function defining the capabilities (facil-
    ities) accessible) to the subject;

ocaps: O -> POWERSET(C), a function defining the capabilities a sub-
    ject must possess in order to observe the object;

mcaps: O -> POWERSET(C), a function defining the capabilities a sub-
    ject must possess in order to modify the object;

icaps: S -> POWERSET(C), a function defining the capabilities a sub-
    ject must possess in order to invoke another subject.

The access relations for the three abstract access modes are
defined by the following axioms:

61

(AI.1) $\forall$ s$\in$S, o$\in$O s $\underline{o}$ o $\Rightarrow$ $\underline{ocaps}$(o) $\subseteq$ $\underline{caps}$(s).

The enforcement of AI.1 requires a subject to have at least an object's observe capabilities in order to observe the object.

(AI.2) $\forall$ s$\in$s, o$\in$O s $\underline{m}$ o $\Rightarrow$ $\underline{mcaps}$(o) $\subseteq$ $\underline{caps}$(s).

The enforcement of AI.2 requires a subject to have at least an object's modify capabilities in order to modify the object.

(AI.3) $\forall$ s[1],s[2]$\in$S s[1] $\underline{i}$ s[2] $\Rightarrow$ $\underline{icaps}$(s[2]) $\subseteq$ $\underline{caps}$(s[1]).

The enforcement of AI.3 requires an invoking subject have at least the invoked subject's invocation capabilities in order to successfully invoke.

We note the similarity of the capability policy to a "lock and key" protection policy. The ability to access an element (subject or object) is determined by the possession of a key that fits the lock of the element. A key may be a "master", able to unlock all locks having subsets of its capabilities.

This policy can be used with the discretionary ring policy (see Section III) to solve the posed protection problem. We partition the set of kernel subjects and objects into categories based on the special privileges they provide. For example, one such category might identify the privileges necessary for the system root process (often the privilege to create processes at any security level). Another important example is the category identifying the kernel privileges associated with ordinary, untrusted processes. We associate a unique capability with each category. The capability to access each category is given only to the facility (trusted process) that requires the special access privileges provided by the category's kernel subjects and objects. The ring policy protects these kernel elements from indiscriminate access (see Section IV) and thus protects their integrity.

62

REFERENCES

1. "ESD 1974 Computer Security Developments Summary," MCI-75-1, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Massachusetts, December 1974.

2. D.E. Bell and L.J. LaPadula, "Secure Computer System: Unified Exposition and Multics Interpretation," ESD-TR-75-306, Electronic Systems Division, AFSC, Hanscom AF Base, Bedford, Massachusetts, January 1976.

3. K. G. Walter, W. F. Ogden, W. C. Rounds, F. T. Bradshaw, S. R. Ames, Jr., and D. G. Shumway, "Primitive Models for Computer Security," ESD-TR-74-117, Case Western Reserve University, Cleveland, Ohio, January 1974.

4. J. P. Anderson, "Computer Security Technology Planning Study," ESD-TR-73-51, Volume I, James P. Anderson & Co., Fort Washington, Pennsylvania, October 1972.

5. R. R. Schell, P. J. Downey, and G. J. Popek, "Preliminary Notes on the Design of Secure Military Computer Systems," MCI-73-1, Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, Massachusetts, January 1973.

6. W.L. Schiller, "The Design and Specification of a Security Kernel for the PDP-11/45," ESD-TR-75-69, Electronic Systems Division, AFSC, Hanscom AF Base, Bedford, Massachusetts, May 1975.

7. E.I. Organick, The Multics System: An Examination of Its Structure, MIT Press, Cambridge, Massachusetts, 1972.

8. Department of Defense, "Security Requirements for Automatic Data Processing (ADP) Systems," Department of Defense Manual 5200.28M, December 1972

9. M. D. Schroeder, "Cooperation of Mutually Suspicious Subsystems in a Computer Utility," MAC-TR-104, MIT Project MAC, Cambridge, Massachusetts, September 1972.

10. J. H. Saltzer, "Protection and the Control of Information in Multics," Communications of the ACM, Volume 17, Number 7, July 1974, pp. 388-402.

REFERENCES (Concluded)

11. C. Weissman, "Security Controls in the ADEPT-50 Time-Sharing System," _Proceedings AFIPS 1969 FJCC_, AFIPS Press, Montvale, New Jersey, 1969, pp. 119-133.

12. M. D. Schroeder and J. H. Saltzer, "A Hardware Architecture for Implementing Protection Rings," _Communications of the ACM_, Volume 15, Number 3, March 1972, pp. 157-170.

13. D.E. Bell and E.L. Burke, "A Software Validation Technique for Certification: The Methodology," ESD-TR-75-54, Volume I, Electronic Systems Division, AFSC, Hanscom AF Base, Bedford, Massachusetts, April 1975.

14. M.A. Harrison, W.L. Ruzzo, and J.D. Ullman, "On Protection in Operating Systems," ACM SIGOPS Operating Systems Review, Volume 9, Number 5, Proceedings of the Fifth Symposium on Operating Systems Principles, pp. 14-25.

15. J.K. Millen, "Security Kernel Validation in Practice," _Communications of the ACM_, May 1976.