C TECHNICAL REPORT 79-91
Library No. S-237,254
(IDA PAPER P-2316)
September 1991

INTEGRITY IN AUTOMATED INFORMATION SYSTEMS

Prepared for
National Computer Security Center (NCSC)
by

Terry Mayfield
J. Eric Roskos
Stephen R. Welke
John M. Boone

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

FOREWORD

This NCSC Technical Report, ``Integrity in Automated Information Systems,'' is issued by the National Computer Security Center (NCSC) under the authority of and in accordance with Department of Defense (DoD) Directive 5215.1, ``Computer Security Evaluation Center.'' This Publication contains technical observations, opinions, and evidence prepared for individuals involved with computer security.

Recommendations for revision to this publication are encouraged and will be reviewed periodically by the NCSC. Address all proposals for revision through appropriate channels to:

National Computer Security Center

9800 Savage Road

Fort George G. Meade, MD 20755-6000

Attention: Chief, Standards, Criteria & Guidelines Division

Reviewed by:________________________________ September 1991

RON S. ROSS, LTC (USA)

Chief, Standards, Criteria & Guidelines Division

Released by:________________________________ September 1991

THOMAS R. MALARKEY

Chief, Office of Computer Security Publications and Support

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

As public, private, and defense sectors of our society have become increasingly dependent on widely used interconnected computers for carrying out critical as well as more mundane tasks, integrity of these systems and their data has become a significant concern. The purpose of this paper is not to motivate people to recognize the need for integrity, but rather to motivate the use of what we know about integrity and to stimulate more interest in research to standardize integrity properties of systems.

For some time, both integrity and confidentiality have been regarded as inherent parts of information security. However, in the past, more emphasis has been placed on the standardization of confidentiality properties of computer systems. This paper shows that there is a significant amount of information available about integrity and integrity mechanisms, and that such information can be beneficial in starting to formulate standardizing criteria. We have gone beyond the definition of integrity and provided material that will be useful to system designers, criteria developers, and those individuals trying to gain a better understanding of the concepts of data and systems integrity. This paper provides foundational material to continue the efforts toward developing criteria for building products that preserve and promote integrity.

We begin by discussing the difficulty of trying to provide a single definition for the term integrity as it applies to data and systems. Integrity implies meeting a set of defined expectations. We want a system that protects itself and its data from unauthorized or inappropriate actions, and performs in its environment in accordance with its users' expectations. We also expect internal data and any transformations of that data to maintain a correct, complete and consistent correspondence to itself and to what it represents in the external environment. Addressing these multiple views in a single definition is difficult. We conclude that a single definition is not needed. An operational definition, or framework, that encompasses various views of the issue seems more appropriate. The resulting framework provides a means to address both data and systems integrity and to gain an understanding of important principles that underlie integrity. It provides a context for examining integrity preserving mechanisms and for understanding the integrity elements that need to be included in system security policies.

We extract a set of fundamental principles related to integrity. These are based on our framework, a review of various written material on the topic of integrity, and an investigation of existing mechanisms deemed to be important to preserving and promoting integrity. These principles underlie the wide variety of both manual and automated mechanisms that are examined. The mechanisms have been categorized to show that they serve a relatively small set of distinct purposes or policies. Some mechanisms that promote integrity are not documented in traditional literature and not all of the mechanisms addressed here are implemented in computer systems. All of these do, however, provide insight into some of the controls necessary and the types of threats that automated integrity mechanisms must counter. We also provide an overview of several models and model implementations (paper studies) of integrity. These models are still rather primitive with respect to the range of coverage suggested by examining both data and systems integrity. The model we found to be receiving the most attention at this time is the Clerk-Lesion Model. Although this is not a formal mathematical model, it provides a fresh and useful point of departure for examining issues of integrity.

From this study, we conclude that it is possible to begin to standardize data and systems integrity properties. Principles exist, trial policies can be formulated and modelled, and mechanisms can be applied at various layers of abstraction within a system. The Institute for Defense Analyses (IDA) has initiated a follow-on study to look at the allocation and layering of mechanisms. We also conclude that there are gaps in our information and that the standardization process could help guide certain studies. Such studies should include the analysis of existing interfaces and protocols to determine the appropriate integrity interfaces or the need to design new protocols. Other demonstration/validation studies should be conducted to show that mechanisms are workable, interfaces are well understood, protocol concepts are valid, and standardized criteria are testable. We conclude that criteria development efforts can occur concurrently with the protocol and demonstration/validation studies.

ACKNOWLEDGMENTS

The National Computer Security Center extends special recognition to the principle authors from the Institute for Defense Analyses (IDA): Terry Mayfield (Task Leader), Dr. J. Eric Roskos, Stephen R. Welke, John M. Boone, and Catherine W. McDonald, as well as the Project Leader (NSA C81), Maj. Melvin De Vilbiss (USA).

We wish to thank the external reviewers who provided technical comments and suggestions to earlier versions of this report. Their contributions have caused this document to evolve significantly from the original efforts. We wish also to express appreciation to the principle reviewers at IDA, Dr. Karen Gordon and Dr. Cy Ardoin, for their technical support. A special thanks goes to Katydean Price for her tremendous editorial support during the course of this project.

The principle authors have dedicated this document in memory of their close friend, Dr. J. Eric Roskos-a talented computer scientist and colleague who performed much of the original research for this effort. His tragic death left a tremendous gap in the research team. Eric is often thought of and very much missed.

1 INTRODUCTION

1.1 PURPOSE

This paper provides a framework for examining integrity in computing and an analytical survey of techniques that have potential to promote and preserve computer system and data integrity. It is intended to be used as a general foundation for further investigations into integrity and a focus for debate on those aspects of integrity related to computer and automated information systems (AISs).

One of the specific further investigations is the development and evolution of product evaluation criteria to assist the U.S. Government in the acquisition of systems that incorporate integrity preserving mechanisms. These criteria also will help guide computer system vendors in producing systems that can be evaluated in terms of protection features and assurance measures needed to ascertain a degree of trust in the product's ability to promote and preserve system and data integrity. In support of this criteria investigation, we have provided a separate document [Mayfield 1991] that offers potential modifications to the Control Objectives contained in the Trusted Computer System Evaluation Criteria (TCSEC), DOD 5200.28-STD [DOD 1985]. The modifications extend the statements of the control objectives to encompass data and systems integrity; specific criteria remain as future work.

1.2 BACKGROUND

Integrity and confidentiality are inherent parts of information security (INFOSEC). Confidentiality, however, is addressed in greater detail than integrity by evaluation criteria such as the TCSEC. The emphasis on confidentiality has resulted in a significant effort at standardizing confidentiality properties of systems, without an equivalent effort on integrity. However, this lack of standardization effort does not mean that there is a complete lack of mechanisms for or understanding of integrity in computing systems. A modicum of both exists. Indeed, many well-understood protection mechanisms initially designed to preserve integrity have been adopted as standards for preserving confidentiality. What has not been accomplished is the coherent articulation of requirements and implementation specifications so that integrity property standardization can evolve. There is a need now to put a significant effort on standardizing integrity properties of systems. This paper provides a starting point.

The original impetus for this paper derives from an examination of computer security requirements for military tactical and embedded computer systems, during which the need for integrity criteria for military systems became apparent. As the military has grown dependent on complex, highly interconnected computer systems, issues of integrity have become increasingly important. In many cases, the risks related to disclosure of information, particularly volatile information which is to be used as soon as it is issued, may be small. On the other hand, if this information is modified between the time it is originated and the time it is used (e.g., weapons actions based upon it are initiated), the modified information may cause desired actions to result in failure (e.g., missiles on the wrong target). When one considers the potential loss or damage to lives, equipment, or military operations that could result when the integrity of a military computer system is violated, it becomes more apparent why the integrity of military computer systems can be seen to be at least as important as confidentiality.

There are many systems in which integrity may be deemed more important than confidentiality (e.g., educational record systems, flight-reservation systems, medical records systems, financial systems, insurance systems, personnel systems). While it is important in many cases that the confidentiality of information in these types of systems be preserved, it is of crucial importance that this information not be tampered with or modified in unauthorized ways. Also included in this categorization of systems are embedded computer systems. These systems are components incorporated to perform one or more specific (usually control) functions within a larger system. They present a more unique aspect of the importance of integrity as they may often have little or no human interface to aid in providing for correct systems operation. Embedded computer systems are not restricted to military weapons systems. Commercial examples include anti-lock braking systems, aircraft avionics, automated milling machines, radiology imaging equipment, and robotic actuator control systems.

Integrity can be viewed not only in the context of relative importance but also in the historical context of developing protection mechanisms within computer systems. Many protection mechanisms were developed originally to preserve integrity. Only later were they recognized to be equally applicable to preserving confidentiality. One of the earliest concerns was that programs might be able to access memory (either primary memory or secondary memory such as disks) that was not allocated to them. As soon as systems began to allocate resources to more than one program at a time (e.g., multitasking, multiprogramming, and time-sharing), it became necessary to protect the resources allocated to the concurrent execution of routines from accidentally modifying one another. This increased system concurrency led to a form of interleaved sharing of the processor using two or more processor states (e.g., one for problem or user state and a second for control or system state), as well as interrupt, privilege, and protected address spaces implemented in hardware and software. These ``mechanisms'' became the early foundations for ``trusted'' systems, even though they generally began wit the intent of protecting against errors in programs rather than protecting against malicious actions. The mechanisms were aids to help programmers debug their programs and to protect them from their own coding errors. Since these mechanisms were designed to protect against accidents, by themselves or without extensions they offer little protection against malicious attacks.

Recent efforts in addressing integrity have focused primarily on defining and modelling integrity. These efforts have raised the importance of addressing integrity issues and the incompleteness of the TCSEC with respect to integrity. They also have sparked renewed interest in examining what needs to be done to achieve integrity property standardization in computing systems. While a large portion of these efforts has been expended on attempting to define the term integrity, the attempts have not achieved consensus. However, many of these definitions point toward a body of concepts that can be encompassed by the term integrity. This paper takes one step further in that it not only proposes an operational definition of integrity, but also provides material for moving ahead without consensus. This is done through an examination of various integrity principles, mechanisms, and the policies that they support as well as an examination of a set of integrity models and model implementations

1.3 SCOPE

Our examination of integrity takes several viewpoints. We begin in Section 2 by looking at the issue of defining integrity. Here we build a framework or operational definition of integrity that will serve our purpose in analyzing mechanisms that provide integrity. This framework is derived from a number of sources, including: (1) what people generally say they mean when they discuss having a system provide integrity, (2) from dictionary definitions, and (3) other writings on the topic that we have interpreted to provide both specific integrity goals and a context for data and system integrity.

In Section 3, we extract a set of fundamental principles from these goals and contextual interpretations. Principles are the underlying basis on which policies and their implementing mechanisms are built. An additional set of basic protection design principles, extracted from Saltzer & Schroeder's tutorial paper, The Protection of Information in Computer Systems [Saltzer 1975], has been provided as an appendix for the convenience of the reader. These design principles apply to the general concept of protection and, thus, are important additional considerations for standardizing integrity preserving properties in computer systems.

Next, in Section 4, we examine a wide variety of manual and automated mechanisms that address various problems related to integrity. Most of these mechanisms, evolving over the course of many years, remain in use today. Several of the mechanisms intended to promote integrity are not documented in traditional computer security literature. Not all of the mechanisms we examine are implemented in computer systems, although they give insight into the types of controls that need to be provided and the types of threats that must be countered by automated integrity mechanisms. Some of the mechanisms we examine appear primarily in embedded systems and others are found in more familiar application environments such as accounting. The mechanisms have been categorized to show that they serve a relatively small set of distinct purposes. We use the term policy to describe the higher-level purpose (categorization) of a mechanism since such a purpose generally reflects administrative courses of action devised to promote or preserve integrity.

Independent of the mechanisms a small number of formal models has been established with differing approaches to capturing integrity semantics. In Section 5, we examine several models that have been proposed in the last decade to address issues of integrity. Several paper studies have suggested implementations of these models as possibilities for real systems. We also look at a number of these model implementations intended to promote or preserve integrity. This examination provides us with a better understanding of the sufficiency of coverage provided by the proposed models and model implementations.

Finally, in Section 6, we present our study conclusions and recommend a set of further studies that should be performed to enhance our understanding of integrity and better enable us to standardize integrity protection properties in systems.

A reference list is provided at the end of the main body; a list of acronyms and a glossary are provided after the appendix.

2 DEFINING INTEGRITY

Integrity is a term that does not have an agreed definition or set of definitions for use within the INFOSEC community. The community's experience to date in trying to define integrity provides ample evidence that it doesn't seem to be profitable to continue to try and force a single consensus definition. Thus, we elect not to debate the merits of one proposed definition over another. Rather, we accept that the definitions generally all point to a single concept termed integrity.

Our position is reinforced when we refer to a dictionary; integrity has multiple definitions [Webster 1988]. Integrity is an abstract noun. As with any abstract noun, integrity derives more concrete meaning from the term(s) to which it is attributed and from the relations of these terms to one another. In this case, we attribute integrity to two separate, although interdependent, terms, i.e., data and systems. Bonyun made a similar observation in discussing the difficulty of arriving at a consensus definition of integrity [Bonyun 1989]. He also recognized the interdependence of the terms systems and data in defining integrity, and submitted the proposition that ``in order to provide any measure of assurance that the integrity of data is preserved, the integrity of the system, as a whole, must be considered.''

Keeping this proposition in mind, we develop a conceptual framework or operational definition which is in large part derived from the mainstream writing on the topic and which we believe provides a clearer focus for this body of information. We start by defining two distinct contexts of integrity in computing systems: data integrity, which concerns the objects being processed, and systems integrity, which concerns the behavior of the computing system in its environment. We then relate these two contexts to a general integrity goal developed from writings on information protection. We reinterpret this general goal into several specific integrity goals. Finally, we establish three conceptual constraints that are important to the discussion of the preservation and promotion of integrity. These definitions, specific goals, and conceptual constraints provide our framework or operational definition of integrity from which we extract integrity principles, analyze integrity mechanisms and the policies they implement, and examine integrity models and model implementations. A diagram of this framework is found in Figure 1 at the end of this section.

2.1 DATA INTEGRITY

Data integrity is what first comes to mind when most people speak of integrity in computer systems. To many, it implies attributes of data such as quality, correctness, authenticity, timeliness, accuracy, and precision. Data integrity is concerned with preserving the meaning of information, with preserving the completeness and consistency of its representations within the system, and with its correspondence to its representations external to the system. It involves the successful and correct operation of both computer hardware and software with respect to data and, where applicable, the correct operations of the users of the computing system, e.g., data entry. Data integrity is of primary concern in AISs that process more than one distinct type of data using the same equipment, or that share more than one distinct group of users. It is of concern in large scale, distributed, and networked processing systems because of the diversity and interaction of information with which such systems must often deal, and because of the potentially large and widespread number of users and system nodes that must interact via such systems.

2.2 SYSTEMS INTEGRITY

Systems integrity is defined here as the successful and correct operation of computing resources. Systems integrity is an overarching concept for computing systems, yet on that has specific implications in embedded systems whose control is dependent on system sensors. Systems integrity is closely related to the domain of fault tolerance. This aspect of integrity often is not included in the traditional discussions of integrity because it involves an aspect of computing, fault tolerance, that is often mistakenly relegated to the hardware level. Systems integrity is only superficially a hardware issue, and is equally applicable to the AIS environment; the embedded system simply has less user-provided fault tolerance. In this context, it also is related closely to the issue of system safety, e.g., the safe operation of an aircraft employing embedded computers to maintain stable flight. In an embedded system, there is usually a much closer connection between the computing machinery and the physical, external environment than in a command and control system or a conventional AIS. The command and control system or conventional AIS often serves to process information for human users to interpret, while the embedded system most often acts in a relatively autonomous sense.

Systems integrity is related to what is traditionally called the denial of service problem. Denial of service covers a broad category of circumstances in which basic system services are denied to the users. However, systems integrity is less concerned with denial of service than with alteration of the ability of the system to perform in a consistent and reliable manner, given an environment in which system design flaws can be exploited to modify the operation of the system by an attacker.

For example, because an embedded system is usually very closely linked to the environment, one of the fundamental, but less familiar, ways in which such an attack can be accomplished is by distorting the system's view of time. This type of attack is nearly identical to a denial-of-service attack that interferes with the scheduling of time-related resources provided by the computing system. However, while denial of service is intended to prevent a user from being able to employ a system function for its intended purpose, time-related attacks on an embedded system can be intended to alter, but not stop, the functioning of a system. System examples of such an attack include the disorientation of a satellite in space or the confusing of a satellite's measurement of the location of targets it is tracking by forcing some part of the system outside of its scheduling design parameters. Similarly, environmental hazards or the use of sensor countermeasures such as flares, smoke, or reflectors can cause embedded systems employing single sensors such as infrared, laser, or radar to operate in unintended ways.

When sensors are used in combination, algorithms often are used to fuse the sensor inputs and provide control decisions to the employing systems. The degree of dependency on a single sensor, the amount of redundancy provided by multiple sensors, the dominance of sensors within the algorithm, and the discontinuity of agreement between sensors are but a few of the key facets in the design of fusion algorithms in embedded systems. It is the potential design flaws in these systems that we are concerned with when viewing systems from the perspective of systems integrity.

2.3 INFORMATION SYSTEM PROTECTION GOALS

Many researchers and practitioners interested in INFOSEC believe that the field is concerned with three overlapping protection goals: confidentiality, integrity, and availability. From a general review of reference material, we have broadly construed these individual goals as having the following meanings:

1. Confidentiality denotes the goal of ensuring that information is protected from improper disclosure.
2. Integrity denotes the goal of ensuring that data has at all times a proper physical representation, is a proper semantic representation of information, and that authorized users and information processing resources perform correct processing operations on it.

3. Availability denotes the goal of ensuring that information and information processing resources both remain readily accessible to their authorized users.

The above integrity goal is complete only with respect to data integrity. It remains incomplete with respect to systems integrity. We extend it to include ensuring that the services and resources composing the processing system are impenetrable to unauthorized users. This extension provides for a more complete categorization of integrity goals, since there is no other category for the protection of information processing resources from unauthorized use, the theft of service problem. It is recognized that this extension represents an overlap of integrity with availability. Embedded systems require one further extension to denote the goal of consistent and correct performance of the system within its external environment.

2.4 INTEGRITY GOALS

Using the goal previously denoted for integrity and the extensions we propose, we reinterpret the general integrity, goal into the following specific goals in what we believe to be the order of increasing difficulty to achieve. None of these goals can be achieved with absolute certainty; some will respond to mechanisms known to provide some degree of assurance and all may require additional risk reduction techniques.

2.4.1 Preventing Unauthorized Users From Making Modifications

This goal addresses both data and system resources. Unauthorized use includes the improper access to the system, its resources and data. Unauthorized modification includes changes to the system, its resources, and changes to the user or system data originally stored including addition or deletion of such data. With respect to user data, this goal is the opposite of the confidentiality requirement: confidentiality places restrictions on information flow out of the stored data, whereas in this goal, integrity places restrictions on information flow into the stored data.

2.4.2 Maintaining Internal and External Consistency

This goal addresses both data and systems. It addresses self-consistency of interdependent data and consistency of data with the real-world environment that the data represents. Replicated and distributed data in a distributed computing system add new complexity to maintaining internal consistency. Fulfilling a requirement for periodic comparison of the internal data with the real-world environment it represents would help to satisfy both the data and systems aspects of this integrity goal. The accuracy of correspondence may require a tolerance that accounts for data input lags or for real-world lags, but such a tolerance must not allow incremental attacks in smaller segments than the tolerated range. Embedded systems that must rely only on their sensors to gain knowledge of the external environment require additional specifications to enable them to internally interpret the externally sensed data in terms of the correctness of their systems behavior in the external world.

It is the addition of overall systems semantics that allows the embedded system to understand the consistency of external data with respect to systems actions.

1. As an example of internal data consistency, a file containing a monthly summary of transactions must be consistent with the transaction records themselves.
2. As an example of external data consistency, inventory records in an accounting system must accurately reflect the inventory of merchandise on hand. This correspondence may require controls on the external items as well as controls on the data representing them, e.g., data entry controls. The accuracy of correspondence may require a tolerance that accounts for data input lags or for inventory in shipment, but not actually received.
3. As an example of systems integrity and its relationship to external consistency, an increasing temperature at a cooling system sensor may be the result of a fault or an attack on the sensor (result: overlooking of the space) or a failure of a cooling system component, e.g., freon leak (result: overheating of the space). In both cases, the automated thermostat (embedded system) could be perceived as having an integrity failure unless it could properly interpret the sensed information in the context of the thermostat's interaction with the rest of the system, and either provide an alert of the external attack or failure, or provide a controlling action to counter the attack or overcome the failure. The essential requirement is that in order to have the system maintain a consistency of performance with its external environment, it must provided with an internal means to interpret and flexibility to adapt to the external environment.

2.4.3 Preventing Authorized Users From Making Improper Modifications

The final goal of integrity is the most abstract, and usually involves risk reduction methods or procedures rather than absolute checks on the part of the system. Preventing improper modifications may involve requirements that ethical principles not be violated; for example, an employee may be authorized to transfer funds to specific company accounts, but should not make fraudulent or arbitrary transfers. It is, in fact, impossible to provide absolute ``integrity'' in this sense, so various mechanisms are usually provided to minimize the risk of this type of integrity violation occurring.

2.5 CONCEPTUAL CONSTRAINTS IMPORTANT TO INTEGRITY

There are three conceptual constraints that are important to the discussion of integrity. The first conceptual constraint has to do with the active entities of a system. We use the term agents to denote users and their surrogates. Here, we relate one of the dictionary definitions [Webster 1988] of integrity, adherence to a code of behavior, to actions of systems and their active agents. The second conceptual constraint has to do with the passive entities or objects of a system. Objects as used here are more general than the storage objects as used in the TCSEC. We relate the states of the system and its objects to a second of Webster's definitions of integrity, wholeness. We show that the constraint relationships between active agents and passive entities are interdependent. We contend that the essence of integrity is in the specification of constraints and execution adherence of the active and passive entities to the specification as the active agent transforms the passive entity. Without specifications, one cannot judge the integrity of an active or passive entity. The third system conceptual constraint deals with the treatment of integrity when there can be no absolute assurance of maintaining integrity. We relate integrity to a fundamental aspect of protection, a strategy of risk reduction. These conceptual constraints, placed in the context of data integrity and systems integrity and the previous discussions on integrity goals, provide the framework for the rest of the paper.

2.5.1 Adherence to a Code of Behavior

Adherence to a code of behavior focuses on the constraints of the active agents under examination. It is important to recognize that agents exist at different layers of abstraction, e.g., the user, the processor, the memory management unit. Thus, the focus on the active agents is to ensure that their actions are sanctioned or constrained so that they cannot exceed established bounds. Any action outside of these bounds, if attempted, must be prevented or detected prior to having a corrupting effect. Further, humans, as active agents, are held accountable for their actions and held liable to sanctions should such actions have a corrupting effect. One set of applied constraints are derived from the expected states of the system or data objects involved in the actions. Thus, the expected behaviors of the system's active agents are conditionally constrained by the results expected in the system's or data object's states. These behavioral constraints may be statically or dynamically conditioned.

For example, consider a processor (an active agent) stepping through an application program (where procedural actions are conditioned or constrained) and arriving at the conditional instruction where the range (a conditional constraint) of a data item is checked. If the program is written with integrity in mind and the data item is ``out of range,'' the forward progress of the processor through the applications program is halted and an error handling program is called to allow the processor to dispatch the error. Further progress in the application program is resumed when the error handling program returns control of the processor back to the application program.

A second set of applied constraints are derived from the temporal domain. These may be thought of as event constraints. Here, the active agent must perform an action or set of actions within a specified bound of time. The actions may be sequenced or concurrent, they may be performance constrained by rates (i.e., actions per unit of time), activity time (e.g., start & stop), elapsed time (e.g., start + 2hrs), and by discrete time (e.g., complete by 1:05 p.m.)

Without a set of specified constraints, there is no ``code of behavior'' to which the active agent must adhere and, thus, the resultant states of data acted upon are unpredictable and potentially corrupt.

2.5.2 Wholeness

Wholeness has both the sense of unimpaired condition (i.e., soundness) and being complete and undivided (i.e., completeness) [Webster 1988]. This aspect of integrity focuses on the incorruptibility of the objects under examination. It is important to recognize that objects exist a different layers of abstraction, e.g., bits, words, segments, packets, messages, programs. Thus, the focus of protection for an object is to ensure that it can only be accessed, operated on, or entered in specified ways and that it otherwise cannot be penetrated and its internals modified or destroyed. The constraints applied are those derived from the expected actions of the system's active agents. There are also constraints derived from the temporal domain. Thus, the expected states of the system or data objects are constrained by the expected actions of the system's active agents.

For example, consider the updating of a relational database with one logical update transaction concurrently competing with another logical update transaction for a portion of the set of data items in the database. The expected actions for each update are based on the constraining concepts of atomicity, i.e., that the actions of a logical transaction shall be complete and that they shall transform each involved individual data item from one unimpaired state to a new unimpaired state, or that they shall have the effect of not carrying out the update at all; servility i.e., the consecutive ordering of all actions in the logical transaction schedule; and mutual exclusion, i.e., exclusive access to a given data item for the purpose of completing the actions of the logical transaction. The use of mechanisms such as dependency ordering, locking, logging, and the two-phase commit protocol enable the actions of the two transactions to complete leaving the database in a complete and consistent state.

2.5.3 Risk Reduction

Integrity is constrained by the inability to assure absoluteness. The potential results of actions of an adversarial attack, or the results of the integrity failure of a human or system component place the entire system at risk of corrupted behavior. This risk could include complete system includes relatively assured capabilities provided by protection mechanisms plus measures to reduce the exposure of human, system component, and data to loss of integrity should be pursued. Such a risk reduction strategy could include the following:

a) Containment to construct ``firewalls'' to minimize exposures and opportunities to both authorized and unauthorized individuals, e.g., minimizing, separating, and rotating data, minimizing privileges of individuals, separating responsibilities, and rotating individuals.

b) Monitors to actively observe or oversee human and system actions, to control the progress of the actions, log the actions for later review, and/or alert other authorities of inappropriate action.

c) Sanctions to apply a higher risk (e.g., fines, loss of job, loss of professional license, prison sentence) to the individual as compared to the potential gain from attempting, conducting, or completing an unauthorized act.

d) Fault tolerance via redundancy, e.g., databases to preserve data or processors to preserve continued operation in an acknowledged environment of faults. Contingency or backup operational sites are another form of redundancy. Note: layered protection, or protection in depth, is a form of redundancy to reduce dependency on the impenetrability of a single protection perimeter.
e) Insurance to replace the objects or their value should they be lost or damaged, e.g., fire insurance, theft insurance, and liability insurance.

(Figure 1. Not available for electronic version.)

Figure 1. Integrity Framework

3 INTEGRITY PRINCIPLES

``There is a large body of principles from among which those pertinent to any application environment can be selected for incorporation into specific policy statements. There is a need to identify as many as possible of those principles as might be of sufficiently general benefit to warrant their inclusion in a list of such principles from which the formulators of policy can select, cafeteria-style, those appropriate to their needs'' [Courtney 1989].

In this section we discuss important underlying principles that can be used in the design of integrity policies and their supporting or implementing mechanisms. These principles involve not only those that we believe are fundamental to integrity, but also those which underlie risk reduction with respect to integrity. These principles were developed from a review of various written material on the topic of integrity, from our framework formulated in the previous section, and by an investigation of existing mechanisms deemed to be important to preserving and promoting integrity.

3.1 IDENTITY

The principle of identity is fundamental to integrity in that it defines ``sameness in all that constitutes the objective reality of a thing: oneness; and is the distinguishing character of a thing: individuality'' [Webster 1988]. Identity allows one to distinguish and name or designate an entity. It is through identity that relationships are attributed and named. It is through identity that functions are distinguished and named.

Identification of users, programs, objects, and resources includes both their classification, i.e., their membership in classes of entities that will be treated in the same or similar manner, and their individuation, i.e., their uniqueness that will allow the individual entities to be addressed separately. It is through the process of identity that one can establish the specification of wholeness and a specification of behavior.

All protected systems requiring authorization and accountability of individuals depend on the unique identification of an individual human user. User identities need to be protected from being assumed by others. User identities need to be authenticated to confirm that the claimed identity has been validated by a specific protocol executed between the system and the unique user. Further, to ensure traceability throughout the system, the individual identity must be maintained for its entire period of activity in the system.

Identity, through the use of conventions for naming, attributing, labelling, abstracting, typing, and mapping, can provide for separation and control of access to entities. Objects created within the system may require additional attribution to expand the dimensional scope of their identity to meet specific system objectives such as confidentiality, proof of origin, quality, or timeliness.

Another fundamental dimension of both subject and object identity is the conveyance of identity attributes via the relationships of inheritance or replication. Inheritance relationships include part-whole, parent-child, and type instantiation. Attributes of interest include privileges conveyed by users to other users or to surrogate subjects (processes acting on behalf of users), and authenticity of origin conveyed to object copies. This aspect of identity is important to most identity-based policies for access control, especially with respect to the propagation, review, and revocation of privileges or object copies.

3.2 CONSTRAINTS

The principle of constraints is fundamental to integrity. A constraint denotes the state of an active agent being checked, restricted, or compelled to perform some action. This is central to the conceptual constraint of adherence to a code of behavior-or to what others have termed ``expected behavior.'' Constraints establish the bounds of (integrity) actions. When viewed from the context of objects, constraints are the transformation restrictions or limitations that apply in transforming an object from an initial state to a new specified (constrained) state. Constraints establish the bounds of (integrity) states.

3.3 OBLIGATION

The binding, constraining, or commitment of an individual or an active agent to a course of action denotes the principle of obligation. Obligation is another fundamental principle of integrity. It is reflected in the terms duty (required tasks, conduct, service, and functions that constitute what one must do and the manner in which it shall be done) and responsibility (being answerable for what one does). The bound course of action, or constraint set, is generally interpreted as always being required or mandatory and not releasable until the course of action comes to a natural conclusion or specified condition. However, the sense of obligation is lost should the individual or active agent become corrupted, i.e., the binding is broken rather than released. In this sense, an active agent within a system, once initiated, is bound to proceed in its specified actions until it reaches a natural or specified termination point or until the state of the system reaches a failure or corruption point that drives the active agent away from the course of action to which it is bound. This failure or corruption point could be the result of an individual yielding to the temptation to perform an unauthorized action either alone or in collusion with others. It also could be the result of faulty contact with the external environment (e.g., undetected input error at a sensor), loss of support in the internal environment (e.g., hardware failure), contact with corrupted objects (e.g., previously undetected erroneous states), or contact with another corrupted active agent (e.g., improper versioning in the runtime library).

There is also a temporal dimension to the course of action to which an active agent becomes bound. This dimension binds sequencing, sets deadlines, and establishes bounds of performance for the active agent. Obligation is then thought of in terms of initiation or completion timing, e.g., eventually starting or completing, beginning or finishing within an elapsed time, initiating or ending at a specified clock time, initiating or completing in time for a new course of action to begin, or completing a specified number of action cycles in a specified time. System designers, especially those involved in real-time or deadline-driven systems, use the temporal dimension of obligation to develop time slices for concurrent processes. Significant obligation issues in time slicing include interprocess communication synchronization and the access of concurrent processes to shared data.

One example of obligation is the concept of protocols, which are obligatory conventions or courses of action for external and/or internal active entities to follow in interacting with one another. Protocols can constrain the states of data or information to be exchanged, a sequence of actions, or the mutual exclusion or synchronization of concurrent asynchronous actions sharing resources or data objects.

3.4 ACCOUNTABILITY

Integrity, from the social and moral sense, implies that an individual has an obligation to fulfill and that the individual is answerable to a higher (legal or moral) authority who may impose sanctions on the individual who fails to adhere to the specified code of action. Holding the individual answerable is the principle of accountability, from which requirements are derived to uniquely identify and authenticate the individual, to authorize his actions within the system, to establish a historical track or account of these actions and their effects, and to monitor or audit this historical account for deviations from the specified code of action. The enforcement strength of sanctions may impact some individuals more than others; simply a reminder of what is expected and the consequences of not meeting those expectations may prove useful in promoting and preserving integrity.

3.5 AUTHORIZATION

One aspect of binding the active entity to a course of action is that of authorization. In essence, authorization is the right, privilege, or freedom granted by one in authority upon another individual to act on behalf of the authority. Employing the principle of authorization provides one means of distinguishing those actions that are allowed from those which are not. The authority may be the leader of an organization, an administrator acting on behalf of that leader, or the owner of a particular asset who may grant another individual access to that asset. The authority may not only grant access to a particular asset, but may also prescribe a specific set of constrained actions that ensue from the access authorization. Thus, there is a binding between the individual, the course of action, and the asset(s) to be acted upon. Attempting to perform outside of these privilege bounds without additional authority is an integrity violation.

Authorizations may be granted for a particular action or for a period of time; similarly, authorization may be revoked. Authorized actions may be further constrained by attributes of the authority, the recipient, and the object to be acted upon. For example, in many systems, the creator of a data object becomes its owner gaining discretionary authority to grant access, revoke granted accesses, and restrict modes of access to that data object. Such access authorization is identity based. However, access to that object may be constrained by certain of its attributes (identified by labels). These constraints may reflect an augmenting rules-based access policy that mandatory checking of corresponding attributes in the individual be accomplished in accordance with specified rules prior to completing the access authorization. These attributes could include National Security Classification Markings, other organizational sensitivity hierarchies or compartmentation, or label attributes related to the quality, e.g., lower quality, ``initial draft,'' associated with document transcribers vs higher quality, ``final edited draft,'' associated with document editors.

There may be a requirement in certain systems to provide for the dynamic enabling or overriding of authorizations. Whether or not the conditions for enabling or override are to be predetermined or left to the judgement of the user, explicit procedures or specific accountable action to invoke an enabling or bypass mechanism should be provided.

3.6 LEAST PRIVILEGE

Privileges are legal rights granted to an individual, role, or subject acting on the behalf of a user that enable the holder of those rights to act in the system within the bounds of those rights. The question then becomes how to assign the set of system privileges to the aggregates of functions or duties that correspond to a role of a user or subject acting on behalf of the user. The principle of least privilege provides the guidance for such assignment. Essentially, the guidance is that the active entity should operate using the minimal set of privileges necessary to complete the job. The purpose of least privilege is to avoid giving an individual the ability to perform unnecessary (and potentially harmful) actions merely as a side-effect of granting the ability to perform desired functions. Least privilege provides a rationale for where to install the separation boundaries that are to be provided by various protection mechanisms.

Least privilege will allow one individual to have different levels of privilege at different times, depending on the role and/or task being performed. It also can have the effect of explicitly prohibiting any one individual from performing another individual's duties. It is a policy matter as to whether additional privileges are ``harmless'' and thus can be granted anyway. It must be recognized that in some environments and with some privileges, restricting the privilege because it is nominally unnecessary may inconvenience the user. However, granting of excess privileges that potentially can be exploited to circumvent protection, whether for integrity or confidentiality, should be avoided whenever possible. If excess privileges must be granted, the functions requiring those privileges should be audited to ensure accountability for execution of those functions.

It is important that privileges and accesses not persist beyond the time that they are required for performance of duties. This aspect of least privilege is often referred to as timely revocation of trust. Revocation of privileges can be a rather complex issue when it involves a subject currently acting on an object or who has made a copy of the object and placed it in the subject's own address space.

3.7 SEPARATION

Separation refers to an intervening space established by the act of setting or keeping something apart, making a distinction between things, or dividing something into constituent parts. The principle of separation is employed to preserve the wholeness of objects and a subject's adherence to a code of behavior. It is necessary to prevent objects from colliding or interfering with one another and to prevent actions of active agents from interfering or colluding with one another. Further, it is necessary to ensure that objects and active agents maintain a correspondence to one another so that the actions of one agent cannot effect the states of objects to which that agent should not have correspondence, and so that the states of objects cannot affect the actions of agents to which they should not have correspondence.

One example of separation is the concept of encapsulation, which is the surrounding of a set of data, resources, or operations by an apparent shield to provide isolation, (e.g., isolation from interference or unspecified access). With encapsulation, the protection perimeter has well-defined (often guarded) entry and exit points (interfaces) for those entities which have specified access. Encapsulation, when applied in the context of software engineering, generally incorporates other separation concepts associated with principles of software design, e.g., modularity and information hiding, and employs the mechanism of abstract data types found in many modern programming languages.

Other separation concepts include time or spatial multiplexing of shared resources, naming distinctions via disjunctive set operators (categorical or taxonomic classification, functional decomposition, hierarchical decomposition), and levels of indirection (virtual mapping). All these separation concepts can be supported by the incorporation of the principle of least privilege.

3.8 MONITORING

The ability to achieve an awareness of a condition or situation, to track the status of an action, or to assist in the regulation of conditions or actions is the essence of the principle of monitoring. Conceptually, monitoring combines the notion of surveillance with those of interpretation and response. This ability requires a receiver to have continuous or discrete access to specified source data through appropriate forms of sensors. It also requires a specification of the condition, situation, event, or sequence of events that is to be checked, observed, or regulated, and a specification of the response that should be provided. This response specification generally includes invocation linkages to alarms and to a family of handler processes, such as resource or device handlers and exception or error handling processes. In some cases, monitors will require more privilege than other subjects within the system.

The principle of monitoring is key to enforcement of constrained actions in that the actions must be observed, understood, and forced to comply with the imposed constraints. When the actions are not compliant, either additional system-provided corrective actions or alarms to request external corrective actions are invoked.

The principle of monitoring is used in mutual exclusion schemes for concurrent processes sharing data or resources (e.g., Hoare's Monitors) and in the operation of interprocess communications of asynchronous processes to provide process synchronization. Monitoring is the basis for auditing and for intrusion detection. Other examples employing this principle include range, value, or attribute checking mechanisms in the operating system, database management systems (DBMS), or in an applications program; an embedded feedback-loop control system, such as a thermostat-driven cooling system; and the security ``reference'' monitor in trusted systems.

3.9 ALARMS

Whenever systems encounter an error or exception condition that might cause the system to behave incorrectly with respect to the environment (an integrity failure), the system designer should incorporate the principle of alarms to alert the human operator or individuals in the external environment to the unmanageable condition. This fact mandates a careful analysis of not only the internal aspects of system, but also an analysis of possible influences from the external environment. Further, the designer must not only consider the alarms, but also their sufficiency.

Alarms must be designed such that they are sufficient to handle all possible alarm conditions. For example, if a small field on a display is allocated to displaying all alarm conditions, and only one alarm condition may be displayed at once, a minor alarm (such as a low-power alarm) may hide a major alarm (such as indication of intrusion). Thus, if an intruder could artificially generate a low-power condition, he could hide the alarm indicating an unauthorized access.

Alarm sufficiency is a technical design issue which, if overlooked, can have serious impact. It must be required that alarms not be able to mask one another. While there may not be room for all alarm messages to be displayed at once, an indicator of the distinct alarm conditions must be given so that the user does not mistakenly believe that an ``alarm present'' indicator refers to a less severe condition than the alarm actually involved. In general, a single indicator should not group several events under the same alarm message. The central concepts here are that alarms must always reflect an accurate indication of the true status of events and alarm messages must always be visible.

3.10 NON-REVERSIBLE ACTIONS

Non-reversible actions can prevent the effect of an action from later being hidden or undone. Non-reversible actions support the principle of accountability as well as address a unique set of problems, i.e., emergency revocations or emergency destruction. Non-reversible actions are in general, simply a type of restriction on privilege. Thus, the principle can often be implemented using mechanisms intended for granting privileges. For example, a non-reversible write operation can be provided by giving a user write access but no other access to an object. Likewise, an emergency destruction operation can be provided, at least in the abstract, by giving a user ``destroy'' permission but not ``create'' permission on an object.

``Write-once'' media provide one example of the use of this principle. These media are useful when the integrity concern is that the users not be able to later modify data they have created. Creation of audit records is another example employing this principle in which users may be allowed to write data, but then not modify the written data to prevent users from erasing evidence of their actions. Disposable locks used on shipping containers (which can only be locked once and cannot be reused) are yet another example of this principle's use.

3.11 REVERSIBLE ACTIONS

The ability to recognize an erroneous action or condition that would corrupt the system if actions that depend on the erroneous conditional state were allowed to continue often establishes the need to back out the erroneous action or ``undo'' the condition. This is the principle of reversible actions. System designers most often incorporate this principle at the user interface, e.g., in text editors, where a user may readily notice keying errors or command errors and reverse them prior to their having a detrimental and not easily reversible or non-reversible effect on the object state. This principle is also used to support atomicity in database transaction processing through the protocol of ``rulebook,'' which undoes the portion of a transaction already accomplished when the entire transaction cannot be accomplished. Such reversible actions are key to leaving the database in a complete and unimpaired state.

3.12 REDUNDANCY

Redundancy in computer systems is a risk-reducing principle that involves the duplication of hardware, software, information, or time to detect the failure of a single duplicate component and to continue to obtain correct results despite the failure [Johnson 1989]. Redundant processing is commonly used in fault-tolerance applications. The same processing is performed by more than one process, and the results are compared to ensure that they match. The need for redundancy varies depending on the application. Redundant processing is commonly used in the implementation of critical systems in which a need for high reliability exists. Examples include multiply redundant processors in avionics systems, and traditional accounting systems in which auditors reproduce the results of accountants to verify the correctness of their results. In situations where a system may be subjected to adverse conditions, such as on the battlefield or hazardous environment, or in systems which may be subject to an adversarial attack that is attempting to disable operations controlled by the system, redundancy may be essential. Thus, it may be desirable to require it for certain systems.

Hardware redundancy is the most familiar type of redundancy, and involves duplicating hardware components. Software redundancy involves adding software beyond what is necessary for basic operation to check that the basic operations being performed are correct. N-version programming in which different teams provide unique versions of the same application program vs replicated versions is one example of software redundancy. The efficacy of software redundancy to support correct operations remains an open issue. For example, it has been shown that n-version programming teams tend to have difficulty with the identical hard problems of an application [Knight 1986].

Information redundancy involves duplication of information. Duplicate copies of information are maintained and/or processed, so that failures can be detected by comparing the duplicated information. To further assist in detection of failures, the two copies of information may be represented in different ways, (e.g., parity bits or cyclic redundancy codes). By exchanging bit positions of individual data bits in a byte or word, or by complementing the bits of all data, failures such as those that modify a specific bit position in a byte or word, or which force specific bits to always be zero or one, can be detected.

Time redundancy involves repeating an operation at several separated points in time, (e.g., resenting a message that was transmitted with errors). While this approach will not detect constant, persistent failures that always cause an operation to fail in the same way, it can often detect intermittent or transient failures that only affect a subset of the repeated operations.

3.13 MINIMIZATION

Minimization is a risk-reducing principle that supports integrity by containing the exposure of data or limiting opportunities to violate integrity. It is applicable to the data that must be changed (variable minimization and the more general case, data minimization), to the value of information contained in a single location in the system (target value minimization), to the access time a user has to the system or specific data (access time minimization), and to the vulnerabilities of scheduling (scheduling regularity minimization). Each application is discussed in more detail in the following sections.

3.13.1 Variable Minimization

The ability of a subject to violate integrity is limited to that data to which a subject has access. Thus, limiting the number of variables which the user is allowed to change can be used to reduce opportunities for unauthorized modification or manipulation of a system. This principle of variable minimization is analogous to least privilege. Least privilege is usually used to describe restrictions on actions a subject is allowed to perform, while variable minimization involves limiting the number of changeable data to which a subject has access.

For example, a subject may be authorized to transmit messages via a communications system, but the messages may be in a fixed format, or limited to a small number of fixed messages in which the subject can fill in only specific fields. Thus, a subject may be allowed to say ``fire type X missile on coordinates __, __'' but may not be allowed to substitute missile type Y for missile type X.

3.13.2 Data Minimization

Variable minimization generalizes to the principle of data minimization, in which the standardized parts of the message or data are replaced by a much shorter code. Thus ``Fire missile'' might be replaced with the digit ``1'', and ``on coordinates'' might be eliminated altogether, giving a message of the form

1 X ___ ___

where ___ ___ is replaced by the coordinates. The shortened forms of standardized messages or phrases are sometimes called brevity codes. When implemented in a computer system, integrity can be further enhanced by providing menu options or function keys by which the operator specifies the standardized message, thus reducing the potential for error in writing the code. On the other hand, as these codes become shorter, there is an increased likelihood of spurious noise or errors generating an unintentional valid message.

3.13.3 Target Value Minimization

The threat of attack on a system can be reduced by minimizing target value. This practice involves minimizing the benefits of attack on a given system, for example, by avoiding storing valuable data on exposed systems when it can be reasonably retrieved from a protected site on an as-needed basis. Distributing functionality among subjects is another means of minimizing target value and, thus, reduces vulnerability. Highly distributed systems use this approach, in which any one processing element is of little importance, and the system is sufficiently widely distributed that access to enough processing elements to have major impact is not feasible.

3.13.4 Access Time Minimization

Access time minimization is a risk-reducing principle that attempts to avoid prolonging access time to specific data or to the system beyond what is needed to carry out requisite functionality. Minimizing access time reduces the opportunity for abuse. Timeouts for inactivity, rotation, and binding of access to ``normal'' or specified working hours are variations of this approach. This principle can serve distinct integrity functions, particularly in the case of more analytically oriented users of data.

3.14 ROUTINE VARIATION

It is desirable to avoid scheduling vulnerabilities, in which time-dependent vulnerabilities can become known, by employing the risk-reducing principle of routine variation. For example, if audits are scheduled at regular times, a bank employee that is capable of successfully subverting other controls may know exactly when to abscond with bank funds allowing sufficient time to cover up the theft or to make a safe getaway. Similarly, if communications are known to occur at specific times, synchronized with a master clock, an adversary is enabled to concentrate efforts for attack on the known periods of activity. The adversary, for instance, could degrade performance of a radio-based network by jamming the frequencies in use at known transmission times.

3.15 ELIMINATION OF CONCEALMENT

If a system contains structures that allow concealment of integrity-violating devices, the risk of undetected access is increased. This fact applies not only to physical concealment (such as places to locate undetected hardware devices), but it applies also to software concealment. Software concealment includes places such as large, poorly structured code that allows Trojan horse or virus code to be inserted without detection, and features of elaborate and complex command languages where ``back door'' command sequences may be hidden without significant risk of accidental or routine discovery during normal usage of the system.

The ``debug'' feature used by the Internet Worm in 1988 is an example of software concealment. ``Debug'' was part of a large electronic-mail program called ``sendmail.'' The code for the ``sendmail'' program was so large and its command structure so complex that few people other than the implementer of the ``debug'' feature knew of its existence. The feature was further concealed by the very limited documentation of the ``sendmail'' program in proportion to its complexity, and by the fact that its command language and syntax were so obscure that guessing the undocumented command was unlikely.

The problem concerning this feature was that the large, poorly structured code of ``sendmail'' made it easy to hide a security vulnerability from systems security personnel performing a systematic search for security vulnerabilities, while not hiding it sufficiently from those who could exploit it. If the proportion of man-hours spent searching for security vulnerabilities during a security evaluation is small in proportion to the number of man-hours spent searching for vulnerabilities in order to exploit them, code whose complexity aids the concealment of security vulnerabilities is more likely to result in an actual breach of security.

3.16 ACCESS DETERRENCE

The risk-reducing principle of access deterrence is employed in mechanisms that discourage rather than prevent access. These mechanisms are useful in certain situations. For example, a system that makes an unpleasant and distracting noise when an unauthorized access attempt is first detected may reduce the ability of an adversary to concentrate on attacking the system. When used on a battery-powered system in the field, such a device could also serve to disable the system by discharging the system's batteries if the deterrent device is not deactivated within a short period of time.

A conventional intrusion alarm siren is an example of an access deterrent. They are a deterrent rather than an absolute mechanism. Devices which produce unpleasant by-products while in operation function similarly, since they discourage tampering with devices during or after operation. An example is the water-activated battery used in some disposable meteorological equipment in the early 1970s. An unpleasant smelling material was left inside of the battery compartment, thus discouraging persons who found the discarded equipment from trying to reactivate the radio transmitter by replacing the battery. Clearly, use of such mechanisms is limited to unusual conditions.

4 INTEGRITY MECHANISMS

In this section, we examine a wide variety of manual and automated mechanisms that address various problems related to integrity. Most of these mechanisms, evolving over the course of many years, remain in use today. Not all of the mechanisms we examine are implemented in computer systems. These non-automated mechanisms were included in our study because they give insight into what types of controls need to be provided and the types of threats which must be countered by automated integrity mechanisms. Also, since it is impossible to predict what functions can or will be automated in the future, we felt that it would be more productive to examine integrity mechanisms in general, rather than to limit our study to currently automated mechanisms. No single mechanism provides the solution to integrity problems, and certain mechanisms may not be appropriate for certain systems. Rather, each mechanism can increase integrity protection for systems that satisfy the assumptions identified in the respective sections.

The mechanisms have been categorized to show that they serve a relatively small set of distinct purposes. We use the term policy to describe the higher-level purpose of a mechanism. In [NCSC 1988], security policy is defined as the set of laws, rules, and practices that regulates how an organization manages, protects, and distributes sensitive information. Our usage of the term ``policy'' is intended to be narrower than its generally accepted usage. We use this term to describe administrative courses of action which characterize a group of mechanisms in promoting or preserving integrity. The relationship among mechanisms grouped under a policy is a common, abstract course of action which addresses one facet of integrity. For example, fault tolerance (discussed in Section 4.4) is an important aspect of integrity protection for a wide array of applications, and can be provided by a variety of mechanisms. Table 1 is a summary of the policies and corresponding mechanisms identified in this chapter.

(TABLE 1 not available for electronic version.)

TABLE 1. Integrity Mechanisms Grouped by Policy and Sub-Policy

4.1 POLICY OF IDENTIFICATION AND AUTHENTICATION

Identification and authentication (I&A) are elements of a supporting policy required for integrity, just as for confidentiality. In order to enforce access controls it is necessary to uniquely identify with confidence who is attempting access. Likewise, when new data is created it is necessary to identify the creator of the data in order to assign any access control attributes based on the data's origin, such as ownership, hierarchical ``quality'' measures, or categories reflecting specialized knowledge on which the data may be based.

The prevalent understanding and discourse on I&A deals primarily with what we have termed ``user I&A.'' However, there are corresponding identification (and authentication) concerns for other entities, such as devices and objects, that must be voiced to address this topic in full generality. Since the underlying principles of user I&A can be applied equally as well to a variety of entities, the traditional view should be extended to include these entities which have not previously been associated with I&A. Thus, while we will emphasize user I&A, we are motivated to include a discussion of I&A for devices and objects as well. Our discussion of this policy is not meant to be a treatise on I&A mechanisms, but rather to highlight some I&A concerns with respect to integrity.

4.1.1 Policy of User Identification and Authentication

There exists a wide range of user I&A mechanisms in use today, including: passwords, biometric devices, behavior analysis, and identification badges. We will not discuss each of these mechanisms separately, as the purposes of all these mechanisms are the same: (1) the identity of an individual needs to be established in order to enforce policies which relate to those individuals, (2) the identification needs to be established with assurance that the individual is whom he claims to be, and (3) the activity of the individual needs to be captured for accountability. Our discussion will instead focus on the general types of mechanisms which are commonly employed and the issues related to the use of each of these types of mechanisms for user I&A.

In order to be effective, I&A mechanisms must uniquely and unforgeably identify an individual. Traditionally, user I&A is based on the use of ``something you know,'' ``something you are'' (variably, ``something you can do''), or ``something you have.'' Both ``something you know'' and ``something you have'' are limited in effectiveness by the fact that they are only associated with a person by possession. A person possesses knowledge or some identifying object, but both of these can be acquired by someone wishing to pose as that individual. Each approach has advantages and disadvantages. What distinguishes these two approaches is how effectively each type of authentication item can be protected.

The principal weakness of ``something you know'' is that it may be duplicated. Not only is it sometimes very easy to learn something someone else knows, but it may be possible to guess it. Because ``something you know'' can be readily retained and reproduced by humans, no special tools, skills, or equipment are generally required to duplicate this type of authentication item. But ease of duplication is also an advantage of this type of authentication item because such information tends to be easily represented to the trusted computing base (TCB) without special equipment. Any authentication data ultimately must be encoded, in some form, in the TCB's authentication database; in this sense, a ``copy'' of the information has to be kept by the TCB in order to be usable in authentication. Since ``something you know'' can usually be directly represented as a character string, it is easy to store for later use by the TCB.

Although easy to copy, if ``something you know'' is genuinely unique, such as a unique nonsense word or number, it may be easier to guard than a physical object. This advantage results from the fact that an item of knowledge normally is fully in the possession of the person it identifies at all times. Unlike a key, card, or other device, ``something you know'' cannot be stolen while temporarily left sitting on a desk, cannot accidentally fall out of a pocket, and in many cases cannot be forcefully acquired unless the person stealing it has a way of verifying at the time that it is correct.

However, poor security practices such as writing one's password down can negate the advantages of using this technique. ``Something you know'' is also inherently vulnerable to interception at its point of entry into the system. The ease of duplication always makes ``something you know'' an imperfect form of authentication, and it is subject to conscientious protection of information for its effectiveness.

By comparison, the major strength of ``something you have'' lies in its difficulty of duplication. ``Something you know'' is, literally speaking, an example of ``something you have,'' so when we speak of ``something you have'' we will by convention mean a physical object rather than an item of knowledge. While such objects require more effort to guard from theft, they can be made using special equipment or procedures that are generally unavailable to the population which poses a threat to the system. The intent is to make duplication of I&A objects too costly with respect to whatever is to be gained by falsifying the I&A objects. This fact discourages duplication, though it does not necessarily prevent duplication by a determined intruder.

The third type of authentication item, ``something you are,'' is much stronger than the first two. After all, the goal of authentication is to verify ``who you are,'' and ``something you are'' is very closely tied to this goal. But, there are also problems with this type of authentication. A major obstacle is the difficulty of building cost-effective peripherals that can obtain a complete enough sample of ``something you are'' to entirely distinguish one individual from another. Cost is also a factor in identifying ``something you have,'' but the authentication item itself can usually be designed to simplify identification by the TCB. ``Something you are'' cannot usually be easily encoded by conventional computer peripherals. It may be relatively easy to build devices that confirm that someone has some distinguishing features, such as weight or finger length. However, the cost of peripherals which have the ability to detect additional detail required to completely distinguish one person from another can be substantially greater.

Fortunately, not everything about a person is required for an adequately unique identification. Specific methods, such as fingerprinting or retinal scans, may be used to reduce costs in comparison to a total examination of all of a person's physical attributes. Yet, even these methods incur greater costs than the use of a password, which requires no additional hardware at all. Furthermore, such methods are not guaranteed to be infallible: identical twins, for instance, would not be distinguishable by DNA readers, and might not be distinguishable by any other specific tests of physical characteristics. In the hypothetical case of entirely identical twins, the two individuals might be solely distinguished by things in the ``something you know'' category.

It may sometimes be useful to distinguish between ``something you are'' and ``something you can do.'' Examples include an individual's speech, writing, or keystroke dynamics, which might uniquely authenticate a particular individual's identity. Obviously the strength of this approach is directly related to how well such a mechanism can distinguish ``unique'' and ``authentic'' behavior. For instance, the mechanism might not be able to distinguish between two very rapid typists, or perhaps a speech-recognition mechanism could be ``spoofed'' by a high fidelity recording of someone's voice.

Other refinements of particular authentication methods may strengthen the mechanism while increasing costs in terms of additional performance or management overhead. For example, the requirement for one-time passwords places some additional constraints on the user and the mechanism, but is very effective in dealing with the problem of reuse or ``playback'' of authentication information. Nonetheless, simple mechanisms (e.g., a ``standard'' password mechanism) can be both effective and efficient in specific applications and environments.

Thus, not only are there cost and feasibility tradeoffs between the various authentication methods, but several methods may be required to give adequate certainty of authentication, which at the most theoretical level is always subject to some amount of error. A common concern for all I&A mechanisms is the degree of assurance which must be assumed regarding the facts that the mechanism is actually communicating with a bona fide user and that such communication has been neither intercepted nor tampered with. Additional measures such as physically securing communication lines and complementing an I&A mechanism with trusted path features can substantially increase the assurance of authentication.

The need for user I&A is essential for integrity when the permitted operations or accesses vary among different individuals. Without identification, it is not possible to know which user a given subject represents, and thus it is not possible to determine access rights. Without authentication, it is not possible to verify that a user's stated identification accurately reflects the user's identity. I&A policies and mechanisms play a fundamental part in supporting most other protection policies. As a result, the protection and integrity of the I&A mechanism(s) and related data is essential.

4.1.2 Policy of Originating Device Identification

One possible extension to I&A is the need for originating device identification. Particularly in the battlefield, where it must be expected that a certain number of devices will be captured, it is essential that it be possible to uniquely identify each device in order to distinguish captured (and thus untrusted) devices from those which are still trusted. Terminal units may be assigned to specific groups in the field, and since the risk of capture of these devices must be considered, unique hardwired identification of each device is desirable. This identification mechanism can be protected against tampering and substitution, for example, by a mechanism that destroys the terminal unit or one of the unit's critical subcomponents (such as encryption keys) if the unit is physically accessed.

The authentication issue is different for device vs human-user identification. A problem with human-user identification is that many of the identification mechanisms are based on something the user knows (an identifier string) which must be further authenticated. In the case of the device with a hardwired identifier, the identifier is part of ``what the device is,'' thus reducing the authentication burden: the concern then focuses on what user is accessing the known device.

4.1.2.1 Mechanism of Device Identification

In traditional systems, devices are uniquely identified primarily to allow them to be separately addressed. In the commercial environment, the need for unique identification has become an issue due to the desire to limit software use to specifically licensed machines. The use of ``serial number'' read-only memories (ROMs) or similar mechanisms is being increasingly used for this purpose.

Where security is concerned, device identification must be unforgeable to the extent possible to prevent a user from easily changing a device to masquerade as another. In a hostile environment, a conventional identifier ROM is not sufficient, since it may be easily replaced. Any identification mechanism that leaves signal and control lines exposed to tampering may allow the enemy to easily modify the identifier, depending on the architecture of the device interface. For example, if data lines beyond the device select logic are exposed, the identifier could be modified by simply grounding or cutting the data lines. If the system bus is exposed and the device addressing mechanism is known, identifier modification can be accomplished with greater difficulty by building a clip-on device that monitors address lines for the ROM address and then drives the bus data lines to the desired identifier values.

For this reason, it is recommended that the data path between the identifier ROM and the hardware that accesses it be protected from access. This path could be protected by locating the ROM on the same integrated circuit (IC) as the related processor circuits, and by isolating the processor from the external bus while the ROM is being read. Similar protections must be used between the time the identifier is read and when it is transmitted to the device or subject requesting the device identifier. At some point in this path, the protection mechanism must shift from physical protection to encryption and related mechanisms because the data will be exposed to modification during transmission to the remote site.

These identification mechanisms, however, do not preclude attack from a sophisticated adversary; they only make it more difficult for the enemy soldier on the battlefield to successfully infiltrate such a system. It is still possible to construct work-alike hardware that allows the device identifier to be changed, and such hardware may be used by the enemy to falsify this identifier as part of an attack.

Because of this vulnerability, device identification should be used to restrict or disable captured devices, rather than to grant privileges or to serve as user identification. Furthermore, it is undesirable to give specific access-control meanings to specific identifier values, other than to distinguish the devices from one another. Despite its limitations, proper design of device identification is essential to giving an increased level of control over devices distributed in the field.

4.1.3 Policy of Object Identification and Authentication

For our purposes, objects are simply passive entities within a system-our use of particular terminology should not be construed to suggest a particular implementation of objects (e.g., object-oriented systems). The topic of object I&A may require a broadening of one's traditional focus on user I&A, and the realization that the principles underlying user I&A may be applied equally to objects. An object's identity can include its class (e.g., classification, type) and a unique name used to reference a particular instance of the class. Similarly, active entities, besides having unique names, can be grouped via clearance level or group membership, thereby forming classes which are meaningful in relation to policy enforcement. An object's class is specified via attributes associated with the object instances, although all attributes of an object need not apply specifically to policy enforcement. Policy can specify proper activity in terms of an (active or passive) entity's name, attributes, or some combination thereof.

In some cases, such principles have been applied in existing mechanisms which enforce policy in relation to objects, but are simply not recognized as I&A principles, perhaps due to an over specialization of the term. For instance, one purpose of user identification is to provide a means to capture the activity of users. However, user activity is usually defined in terms of actions on specific objects within the system. Thus, an object's identity must be established with certainty in order to accurately capture the activity of a user. Alternate naming facilities, such as aliases or symbolic links, must not hinder the system's ability to resolve and record the unique identity of the physical object. Here we see that the same principle (unique identification) must be applied equally to users and to objects, otherwise unauthorized actions might go undetected.

An object's identity provides a means for both the system and users to bind a logical entity with its physical (i.e., machine) representation. As such, an object's identity can also be expanded by attaching contextual attributes (e.g., owner, type, authenticity), allowing the system to achieve a greater degree of functionality in regard to its interactions with objects. In particular, object identity and its associated attributes enable the system to enforce various policies related to specific operations on instances of objects. Policy decisions are based either directly or indirectly (through associated attributes) on an object's identity. The correspondence of a logical entity to its physical representation can make policy enforcement difficult, or may create opportunities to otherwise circumvent policies. This correspondence can be expressed as the unification of a logical entity with its physical representation; in general, this unification must be maintained during all operations on an object, throughout its period of existence on the system.

Problems associated with the unification of information with data result from two causes: physical or logical dispersion of object identity. Physical dispersion exists because there may be multiple ways to physically access the logical entity specified by a unique system identifier. If physical resources can be directly accessed, then protection mechanisms based on object identification can be circumvented. If temporary objects can be accessed, such as a buffer holding the results of a database query or a scratch file on disk, the integrity of other (permanent) objects may be compromised. Also, objects within a system will exist in different forms (i.e., on disk, in cache, in memory) over their lifetime within the system. If all forms of an object are not sufficiently protected, the integrity of the object can eventually be compromised. For instance, a file may be protected by the operating system while in memory, but a channel device which contains maliciously written software may modify the file while transferring it to or from the disk.

While physical dispersion concerns an internal (to the system) binding of an object to an identity, logical dispersion is associated with the binding of an external ``information unit'' to a system object or set of objects. It is important that an object's identity and associated attributes reflect the external expectation of what that object represents. For instance, if duplicate, sensitive information is entered into two different text files on a system, then that information is necessarily represented internally by two different object identities. If the two files are marked with different integrity attributes, then policy decisions may be inconsistent with respect to identical units of information. One of the text files may be marked as Sales and the other as Development, or perhaps they are ``owned'' by different subjects. While, the system may perform flawlessly in enforcing an integrity policy based on these labels or ownership, different actions may be specified, allowed, or disallowed for each object due to the difference in their attribute specifications. Thus, drastically different versions may emerge over time.

The scope of this problem increases when one realizes that the information in one internal object can be copied to other objects. Even if the attributes associated with the (original) object are faithfully transferred to all objects which contain all or part of the original information, each of those ``new'' objects must have different internal identities. While the overall intent of some integrity policy may be to control access to a ``specific instance of information,'' the system may have dispersed internal identities which contain that information. Because a system's enforcement of policy is usually on a per object basis, it may be impossible for that system to enforce (integrity) policies which address the information ``identity,'' rather than internal object identities.

An example exhibiting the dispersion problem is that of updating identical, distributed databases. The databases represent a single, logical ``instance of information,'' yet at any given moment the actual contents of the databases may be different. If the overall system depends on the cooperation and interaction of components, some of which have different values for the same logical entity, the results are unpredictable. The integrity of such a system's operations may be suspect.

The problems associated with dispersion are compounded when one considers that for integrity concerns, the attributes associated with an identity might not need to be identical for all versions of an object, but there may in fact need to be some variation of the original attributes for each copy. Thus, we suspect that a mechanism to address the dispersion problem will have some of the properties of a ``version control'' or configuration management system which binds related system objects to a single system identity.

Another concern associated with object I&A is that the true identity of an object may be known only by considering the totality of that object's representation on the system (e.g., its name and location). The presence of variable search paths for executable objects may result in the unintended invocation of improper programs. On some systems, executables can be referenced by a sequential search for the program through an arbitrary series of file system locations (i.e., directories). This series is arbitrary in the sense that the specific locations and the order in which they occur in the search path are not fixed properties of the system, but may be specified and varied on a subject to subject basis. If two programs have the same name but reside in different directories, the actual program executed through invoking that name would depend on which directory occurred earlier in a subject's search path.

The presence of different forms of referencing an object generally reflects the desire for flexibility in systems. However, such flexibility can result in some uncertainty about which objects are actually referenced by the invocation of identical commands in different environments. It would seem that the definite resolution of object identity must be guaranteed by the system in order to maintain a high assurance of integrity. This implies that some method of absolute referencing be provided by the system, and used exclusively in operations which are integrity-sensitive. Absolute referencing should extend to interactive commands, dynamic references of applications, and system-internal references. However, for universal absolute referencing to be practical, the referencing mechanism must be flexible. The explicit-path method described above may prove too cumbersome; however, flexible alternatives already exist (e.g., capabilities), although these alternatives may have drawbacks of their own.

Object ``authentication'' may often simply mean that its source of origination is genuine the user who created the object has been authenticated. It may be suggested then that object authentication is then equivalent in some sense to user I&A. However, there are some additional constraints which are assumed in such a proposition. In general, nothing associated with an object throughout its lifetime on the system is static.

Its identity can be changed by renaming or making a copy. Attributes associated with an object can be modified and then possibly changed back to the original. For many if not most objects, the contents of that object will change over time. The authenticity of an object created and maintained by a particular user cannot be ascertained if other users are able to modify the object's name, attributes, or contents. Also, an object may be copied and illegitimately presented as original. As such, the authenticity of an object is highly dependent upon a system's capabilities and the proper administration of controls over that object.

In general, object authentication can be addressed to varying degrees, depending on the capabilities and features offered by a particular system. Some systems, such as network and other communications systems, may provide strong authentication of objects through their protocol structures. The primary motivation is to be able to recognize ``valid'' frames and packets, discarding those with errors or possibly rerouting others to their proper address. However, the protocol structure provides a convenient location to insert more rigorous object authentication mechanisms, in general [ISO 1990]. Other systems such as stand-alone, personal computers, may offer no intrinsic object authentication mechanisms. Objects on such systems may be ``authenticated by recognition,'' but no formal protection features are offered.

Some systems may provide general authentication mechanisms such as digital signatures, notarization, or encryption, which are discussed below. While general authentication mechanisms provide the capability to protect specific objects on demand, general authentication of all objects within a system using these methods may entail a prohibitive administrative overhead. The topic of practical object authentication is one requiring more research to arrive at general solutions.

4.1.3.1 Mechanism of Configuration Management

Configuration management (CM) can be thought of as a mechanism which incorporates aspects involving both identification and authentication of objects within a system. In its most widely accepted usage, CM deals with controlling the construction of complex objects, extending to both identifying the components of an object and how those components are combined. Configuration management may incorporate features of access control for greater measures of protection, but can provide unique protection services independently of access control. Protection provided by CM mechanisms can extend to an arbitrary level over each object considered a component of a more complex object.

Configuration management mechanisms are currently in wide use in the software development industry today, aiding in the control of production software. A more general adaptation of CM techniques in the control of related objects seems plausible. It should be noted that an application (i.e., a program or set of programs) will often provide a significant amount of control over objects within its particular domain. Therefore, a more generalized CM mechanism focused on object integrity would need to extend CM controls over inter-application objects and possibly related, system-level objects.

4.1.3.2 Mechanism of Version Control

A mechanism which is usually integrated into CM systems is that of version control. Version control allows distinct versions of an object to be identified and associated with independent attributes in a well-defined manner. For instance, three different versions of a software package could be marked as ``invalid,'' ``distribution,'' and ``next release,'' respectively. Each version can be handled differently with respect to a particular policy. Version control can include such features as the destruction of ``invalid'' objects or distribution of sanitized versions.

4.1.3.3 Mechanism of Notarization

In the context of distributed systems, it has been suggested that the originator of an object should be responsible for assuring its confidentiality, while the recipient of an object should be responsible for assuring its integrity [Jueneman 1989]. From a detection oriented viewpoint, these are valid points. However, when emphasizing the prevention of policy violations, we find that the converse situation must also apply: the recipient of an object must also protect an object's confidentiality, while the originator must provide some protection of an object's integrity. In particular, the authenticity of an object must, in part, be the responsibility of the originator. The originator must be responsible for the proper creation, maintenance, and administration of the object up until the time of its transmission to the recipient. Similarly, the system on which the object resides must provide the proper mechanisms to support the originator's protection of the object.

Given that an object is beyond the ability of the recipient to protect before it is received, the recipient may be forced to assume that the originator intended for that particular object to be sent, and that the object had been appropriately protected while it resided on the originator's system. This assumption can be unfounded for several reasons: (1) it is possible that the originator's intentions are malicious, (2) the authentication mechanism may have been applied improperly, (3) the authentication mechanism itself may contain faults, and (4) it may have been possible to intercept and replay authentic traffic. In this respect, some authentication measures taken by the recipient often only verify that the originator of an object is genuine.

One technique which can strengthen recipient's assurance in the authenticity of an object is the generalized mechanism of notarization. With this mechanism, an intermediary entity intercedes between the originator and recipient during an object transfer, to establish and certify the authenticity of the object. The intermediary, or notary, ``seals'' the object that bears the signature and/or seal of the originator. In its normal usage, notarization primarily serves to verify the authenticity of originator of an object, rather than the object itself. A generalized extension of this technique seems particularly well suited for automated systems. In automated systems, the originator of an object transfer might simply supply a notary entity with the object, while the notary entity would perform the actual application of the authentication mechanism. This technique has the property of separating the ability to both originate and authenticate an object, which would perhaps be desirable for certain systems or applications. Furthermore, a notary entity might supplement the authentication measures taken by the originator, thereby reducing the required degree of trust for both entities.

4.1.3.4 Mechanism of Time Stamps

One measure which can be used either independently or as a strengthening feature of other mechanisms (e.g., notarization) is the mechanism of time stamps. Time stamps can be provided by a system automatically as an integral part of the system's object-transfer mechanism(s). Time stamps can be used to enforce ``time of use'' dependencies, and can provide added assurance that an object received by an entity is still valid. For instance, in a real-time system, stale data from sensors could be detected and discarded by a recipient processor if the time differential was too great for tolerances. Similarly, there exists multiuser systems today which limit the valid ``time of use'' period for sensitive objects via the use of time stamps [Steiner 1988]. Automated features such as time stamps do not completely solve all aspects of the problems associated with object authentication, although they can provide extra protection. The main advantages of time stamps are that they can be made efficient, transparent, and uncircumventable.

4.1.3.5 Mechanism of Encryption

In general, the mechanism of encryption effectively seals the information within an object inside an additional (logical) container. Used primarily to provide confidentiality, general encryption can be used to ensure the detection of integrity violations and to otherwise hinder integrity attacks. Encryption is not absolute protection, as the sealing process may only be as safe as the encryption key. Also, encryption of an object does not in and of itself prevent damage to its integrity. However, encryption does provide an additional level of protection which must be circumvented in order to violate protection policies, or to succeed at making violations without detection. A distinct advantage of encryption is its flexibility of use its ability to be used either as blanket protection or ``on demand,'' and its applicability to a wide array of object types. There is a great deal of existing literature on the various uses and methods of employing encryption which will not be summarized in this paper for the purpose of brevity. However, specific uses of encryption which are the most relevant to the issues of integrity are discussed below.

4.1.3.6 Mechanism of Digital Signatures

The mechanism of digital signatures is intended to produce the same (desired) effect as a real signature: an unforgeable proof of authenticity. In [Rivest 1978] the authors describe a specific implementation of digital signatures in a public key cryptosystem. Every user (and TCB) has a unique, secret decryption procedure D that corresponds to a publicly available encryption procedure E. As an example scenario, suppose that A and B (also known as Alice and Bob) are two users of a public key cryptosystem. Their encryption and decryption procedures can be distinguished with subscripts: EA,DA,EB,DB. If Bob wants to send Alice a ``signed'' message M in a public key cryptosystem, he first computes his ``signature'' S for the message M using DB:

$$S = DB(M)$$

He then encrypts S using EA (for privacy), and sends the result EA(S) to Alice. He need not send M as well; it can be computed from S.

Alice first decrypts the ciphertext with DA to obtain S. She knows who the presumed sender of the signature is (in this case, Bob); this can be given if necessary in plain text attached to S. She then extracts the message with the encryption procedure of the sender, in this case EB (available on the public file):

$$M = EB(S)$$

She now possesses a message-signature pair (M, S) with properties similar to those of a signed paper document. Bob cannot later deny having sent Alice this message, since no one else could have created S = DB(M).

4.2 POLICY OF AUTHORIZED ACTIONS

Users assigned to particular tasks may have certain actions for which they are uniquely authorized. For example, only the approval authority may be authorized to give the order to launch an attack on the enemy. In computer systems, these authorized actions are often called privileges. Authorized actions can be either unconditional or conditional. Unconditional authorization immediately authorizes an action, although the action may not take place until some time in the future. Conditional authorization authorizes an action, but only if certain conditions hold.

When looking at the topic of authorized actions, we are considering ways in which access may be controlled, without implying any underlying strategy (such as strategies for deterrence). There may be situations in which the need for dynamic modification of authorized actions is required. Such modifications are usually performed by an authorized individual such as a security officer, or by a trusted software component. However, there may be cases in which such modifications must be allowed to be made by normally unauthorized individuals under controlled conditions. It is necessary that emergency situations be considered in the design of authorization systems, since survival of the users may require that overrides be possible.

An authorized actions policy specifies which actions a subject is allowed to perform, rather than controls on which data the subject is allowed to access. We examine several mechanisms that can be used to specify authorized actions under the policy of separation of duties. We begin, however, by addressing the policy of conditional authorization.

4.2.1 Policy of Conditional Authorization

If we consider the fully prohibited (disabled) and fully permitted (enabled) conditions to be the two extremes of unconditional authorization, there are intermediate states between these two where conditional authorization can also be important. We will examine two conditional authorization mechanisms: conditional enabling and value checks. Conditional enabling authorizes an action only when certain conditioning events exist, and value checks are used to restrict actions based on authorized values.

4.2.1.1 Mechanism of Conditional Enabling

Conditional enabling disables the ability to perform a given action (such as to fire a weapon) when the action has not been authorized. Attempting to perform the action will produce no effect. However, if conditionally enabled, an authorization override or bypass mechanism is provided. If the user of the disabled device actuates some arming control, for example, the previously disabled function will now operate. It is expected (and enforced by disciplinary measures as appropriate) that the operator of the device will not bypass the required authorization except in situations that necessitate it. The mechanism serves primarily to prevent accidental actions from being initiated. The requirement that the operator take an explicit action to override the normal requirement for authorization serves to ensure that the action was intentional, and places greater reliance on the operator's contribution to the integrity of the system.

An example of this mechanism that is directly related to our topic is that of recently-designed ``fly-by-wire'' aircraft [Seecof 1989]. These aircraft have computer controlled restrictions on which actions the pilot may perform; the computer determines which actions might be damaging to the aircraft if carried out, such as excessively abrupt changes in direction or excessive engine speed. If the pilot attempts to operate a control outside the calculated safe range, the system prevents the control from being moved into that position. However, if the pilot pushes the control with unusual force, the system will allow him to move it beyond the restricted position. This override allows the pilot to exceed calculated safety limits in emergency situations, such as collision evasion. Considerable controversy has surrounded designs of commercial aircraft that have failed to incorporate such overrides, including suggestions by aircraft engineers at competing manufacturers that such aircraft will not perform adequately in emergency situations [Outposts 1989].

For this reason, it seems advisable that support for conditional overrides of this type be available in safety critical systems where the operators are considered sufficiently skilled, or where absence of the override may result in dangers that the operator could otherwise prevent. In designing such an override, individual analysis of the design is necessary to determine whether and to what extent the override should be provided. It is not possible to give rules that will cover all cases adequately without examination of individual cases. Simply omitting to provide support for such overrides, or prohibiting them in standards, appears inadvisable.

4.2.1.2 Mechanism of Value Checks

Value checks are a weak form of type enforcement, and historically are the predecessor to abstract data types and strong typing discussed in Sections 4.3.1.1 and 4.3.1.2. There are several related types of checking techniques that we have grouped under the heading of value checks: discrete value checks, range checks, and data attribute checks.

With discrete value checks, data may be permitted to have only certain values out of a wider possible set, or may be restricted not to have particular values. For example, target coordinates may be restricted such that artillery could not be directed to fire upon friendly forces.

Range checks verify that data is within a certain range of values. For example, memory parameters passed from a memory management process to the kernel must be checked to be certain that they fall within certain bounds. Range checks supported by the machine architecture have generally been limited to those required for correct operation of the instruction set, such as checks for arithmetic overflows, and those required in support of the memory protection subsystem, such as checks to ensure that addresses are not out of range.

Data attribute checks, such as verifying that data is in a particular numeric format, have been present for several decades in the instruction sets of commercial machines. These elementary checks usually involve only verifying that the data is in a format suitable for correct processing by the machine's defined set of operations, e.g., that a packed decimal number consists only of the hexadecimal representation of the digits 0-9 followed by a valid sign digit (A-F) [Struble 1975]. More complex attribute checking is possible through programming language constructs. For example, values may be required to be numeric or alphabetic, numbers may be required to be even or multiples of some other value, and strings may be required to be of a certain length.

Checking mechanisms beyond these have been implemented either at the programming language level, or as a programming practice in applications development. Subscript range checking, for example, has been a debug feature provided by some compilers even when the language involved does not incorporate strong type enforcement, simply because it helped to detect a commonly occurring error. However, such checks were often considered debug-only features, and were usually removed for efficiency reasons in production software.

Newer languages retain these checking mechanisms, but extend them by providing for exception handlers: when an error is detected, a means is provided for the program to retain control and handle the error appropriately. Similar error-handling mechanisms existed in much earlier systems, but generally were a feature of the processor architecture which might on occasion be extended into the operating system interface where it was accessible to the user. They were not well integrated into most programming languages.

4.2.2 Policy of Separation of Duties

Many integrity violations result when a subject fails to perform expected duties in the appropriate way. A violation can involve willful inappropriate actions (e.g., conspiracy, coercion, and duping), failure to perform duties, or performance of duties in inadequate ways. To promote prevention and detection of these types of violations, separation of duties may be employed. In this type of policy, different subjects are given distinct but interrelated tasks, such that failure of one subject to perform as required will be detectable by another. To avoid circumvention of this mechanism, any one subject is explicitly prohibited from performing another subject's duties.

This policy is based partly on the expectation that not all subjects involved in interrelated tasks will attempt integrity violations at the same time (i.e., collaborate), and that those who do not attempt integrity violations will detect the failure of their counterparts. People may be less likely to attempt integrity violations if they know that their actions will be visible to others who are performing related duties. Likewise, people may be more likely to notice and recognize situations that are anomalous and to report them. Depending on how duties are partitioned, separation of duties may altogether prevent a single individual from violating integrity. For example, some tasks may be partitioned among several subjects such that, while none of the subjects can observe their counterparts' actions, the task cannot be completed without a contribution from all the subjects.

It must be noted that the preventive effect may not always be provided by a simple separation of duties. For example, if one individual is allowed to issue checks but not update inventory records, and another is allowed to update inventory records but not issue checks, the first individual may still issue unauthorized checks. The fact that the checks are unauthorized will be detected by auditors who find that the payments did not result in items being added to the inventory, so the fear of being detected is the principal deterrent. The example could be enhanced to include the preventive effect if the duty of issuing checks was further divided to limit a first person to providing a proposed table of check addresses and amounts, and a second person to authorizing the AIS to print the checks and controlling the check stock.

It should also be noted that restricting access may prevent an individual from filling in for another individual in an emergency situation, such as a situation in which the individual originally assigned a given task is injured on the battlefield. Thus, separation of duties also increases the extent to which individuals become critical components in the overall functioning of a system. If other persons are prevented from substituting for the missing person, the system may be disabled more easily by the failure of an individual to perform their assigned duty.

One of the stronger points of separation of duties is that, in certain systems, it is possible to be reasonably certain that not all subjects will fail. For example, it may be expected that an intruder will not be able to find ways to violate the integrity of all subjects among which duties are separated, and thus that the remaining subjects will detect the attempted integrity violations of the other subjects. On the other hand, if this expectation is incorrect, and the intruder finds a way to cause ``collaboration'' between all the subjects involved, this policy will not detect or prevent the violation.

In the case of machine processes, the preventive effect can be achieved when it is possible to partition the functions of the different processes such that a single errant process cannot in itself cause a harmful result to occur. However, the psychological element involving the fear of detection is absent in machine processes. Since they merely perform their operations as programmed, without detailed evaluation of risks, machine processes will not be deterred a priori from committing integrity violations. In the same way, they will only detect violations for which they have been programmed to check, and are not likely to recognize many forms of suspicious behavior by their counterparts. A machine process which is involved in separation of duties with a human counterpart is particularly vulnerable, since the person may find it easy to ``outwit'' the machine's checks, especially if the person comes to understand the machine's checking algorithms.

The enforcement of separation of duties can be either static or dynamic. The simplest approach is static separation, where the system administrator in charge of maintaining access controls is responsible for understanding the way the rules achieve separation of duties and for making appropriate assignments. This approach is limited in functionality because it is often necessary or desirable to reassign people to duties dynamically. An alternative approach is for the system to keep track dynamically of the people who have executed the various actions in the task sequence, and to ensure, for any particular execution, that proper separation has occurred. However, there are several hard problems to solve in order to implement dynamic separation of duties, such as encoding valid sequences and separately tracking each execution of a particular sequence.

4.2.2.1 Mechanism of Rotation of Duties

In some tasks, it is likely that a person performing the task may discover, through random events, ways to compromise the integrity of the system. If such a person is corrupt or corruptible, the probability that the person can cause an integrity violation increases the longer he is assigned to the same duty. The probability that the violation will remain undetected is also increased in certain instances. To address these vulnerabilities, rotation of duties is used, in which a person is assigned to a given task for only a limited amount of time before being replaced by another person.

For example, a person may discover exploitable ``loop-holes'' in a system and may decide to use them to violate the integrity of the system. If the person is rotated out of a given position before such loopholes are discovered, some threat to integrity may be reduced. Note, however, that communication between current and former occupants of the vulnerable role may defeat the protection provided by this mechanism. Rotation may also give the user a broader understanding of the operation of the system, giving more insight into how to circumvent integrity.

Additional features of rotation of duty involve limiting the damage one individual can do and increasing the likelihood of detection. If a given individual is seeking to bring about violations of integrity, rotation can accomplish two things. First, if there is only one task in which an individual has discovered a way to perform an access in violation of policy, rotation will reduce that individual's exposure to the vulnerable task. Second, rotation may enable identification of the individual involved in an integrity violation: if an audit shows that a problem exists only while a given individual is assigned to a particular task, this provides evidence that individual may be associated with the problem in some way. If the individual has discovered ways to violate integrity in multiple duties, an audit may indicate that the problem ``follows'' the individual as rotation occurs, again providing evidence that the individual in question may be involved in the violation.

Rotation of duty also addresses an integrity threat that does not involve willful malicious action: the tendency for errors to increase when a person performs a monotonous task repeatedly. In this case, rotation of duty reduces the amount of time the repeated, monotonous task must be performed by a given person. In general, electronic systems do not have an analogous property, although mechanical systems may (i.e., rotation to distribute mechanical wear among exchangeable parts).

This mechanism implies separation of duties, since in order for duties to be rotated among different users, the duties must first be partitioned among these users. Thus, it is necessary that a mechanism be provided to enforce separation of duties. For rotation of duties, this enforcement must be extended to allow orderly, policy driven rotation of assignments to specific duties. The rotation may occur at a specific point in time, or in response to requests by an administrator.

Rotation of duties can also be used to effectively rotate data. For example, if three accountants are maintaining the accounts of three different companies, the accountants could be rotated among the different companies periodically. The duty of accounting would not change (and thus no additional training would be necessary), but the data being maintained would be rotated among the accountants, reducing the probability that an authorized user would make improper modifications.

4.2.2.2 Mechanism of Supervisory Control

In some cases, authorization may be delegated to subjects in controlled ways. Supervisory control exists when a subject in a supervisory role is required to authorize specific actions to be carried out by a subordinate subject as certain conditions arise. In general, the authorization would apply to exactly one occurrence of an action so that overall control of the activity remains with the supervising subject. Supervisory control may involve either requiring an individual to give final approval to an action being initiated by another individual, or requiring an individual to give final approval only to actions that meet certain constraints (such as those that have particularly severe consequences). Normal activity outside the scope of these constraints would not require the approval of the supervisor.

This mechanism provides for delegation of less integrity critical tasks to less trusted individuals, with only the integrity critical portions being dependent on the supervisor. One example is check cashing authority at grocery checkout stands. A checkout clerk can initiate the check cashing procedure, but must receive final approval from a supervisor for the actual transaction to take place.

Supervisory control is often a complexity-control mechanism. The delegation of action is often a tradeoff, since it might be desirable under ideal circumstances for all actions to be performed by the most trusted individual, but doing so would not be possible due to the number of individual actions that must be decided upon and undertaken. This fact highlights one vulnerability associated with supervisory control if the critical conditions which require a response occur with high frequency, the supervisor may spend an unacceptable portion of time issuing authorizations to subordinates to deal with the occurrences. Therefore, it may be inadvisable to rely on supervisory control if such a high frequency situation is likely to occur, unless this contingency has been planned for (e.g., through the granting of blanket authorizations).

4.2.2.3 Mechanism of N-Person Control

N-person control (where N is a number, typically 2) requires more than one subject to request or participate in the same action in order for the action to be successful. An example of this mechanism is a system for commanding missile launches in which two physically separated controls must be operated by two people simultaneously. The rationale is that n people are less likely to spuriously decide to initiate an action than would one person.

This is a unique mechanism, which is not provided in most existing computer systems. It is roughly analogous in its underlying purpose to the preventive aspect of separation of duties, in that it serves to require that multiple persons cooperate to bring about a controlled action. The difference is that in N-person control, separate individuals are performing the same duty (as opposed to different duties in separation of duties) to bring about a single action. N person control often also requires that these persons perform actions simultaneously; by comparison, separation of duties usually does not involve this requirement for simultaneity. As with may other mechanisms, N-person control can be strengthened by incorporating other security features, such as time tokens, trusted path, and physical security.

4.2.2.4 Mechanism of Process Sequencing

Certain duties may be required to be performed in a certain order; this process sequencing can serve at least two purposes. First, this mechanism can ensure that all steps necessary for a multi-step procedure are followed in the proper order. Second, if the sequence is known only by current authorized users, it may be more difficult for an adversary to successfully discover the sequence and violate the integrity of the system. Process sequencing especially discourages trial-and-error attacks, if the sequence is long and an improper sequence makes it necessary for the user to start over at the beginning. However, it can also make it more difficult for an authorized user to succeed at the required steps, if the procedure or environment is such that errors by users are likely.

4.3 POLICY OF SEPARATION OF RESOURCES

In addition to separating duties among different persons or processes, access to resources can also be separated. In this policy, the resources to which subjects have access are partitioned so that a given subject has access only to a subset of the resources available. More precisely, a given subject and the tools (e.g., programs, system utilities, system applications) available to that subject are allowed access only to specific resources. An example is disk quota enforcement, in which the amount of disk space that each subject may use is limited to a fraction of the total space available in the system. This policy can overlap with the separation of duties policy discussed previously if the resources that are controlled limit the duties that each subject can perform. However, separation of resources can also have other effects, such as limiting the extent of damage to otherwise homogeneous resources that can be caused by a given errant subject.

Similar principles apply to separation of resources as apply to separation of duties. As with separation of duty, a means of partitioning is required, but the partitioning separates resources rather than duties and tools. To accomplish this end, the TCB should provide a means of enforcing an appropriate partitioning of resources such that they are accessible only to persons or processes in specific roles.

To a certain extent, such mechanisms are already provided in existing systems. Identity based access control allows such discretionary partitioning, e.g., owner has discretion to grant access. In some cases such partitioning is ``mandatory.'' For example, in many systems, only the owner of a file is allowed to perform certain operations on it, such as deleting it or changing its access attributes. Permission to change access permissions, identified in [Bishop 1979] as ``grant'' permissions (sometimes inexactly called ``control'' permissions), is ``mandatory'' on such a system because it is not possible for anyone to modify this permission to allow someone else other than the owner of the file to change the access permissions. Thus, the presence of this access control is mandated by the implementation of the system security policy, and cannot be overridden at the discretion of the users. The word ``mandatory'' in this sense has a different meaning than the word ``Mandatory'' in ``Mandatory Access Controls,'' since the latter usage refers to a specific type of access control whose mandatory nature is based on rules that must be followed. Adding further mandatory or rule based restrictions specified on a per user basis could strengthen most identity based, discretionary partitioned systems.

However, there is an additional aspect to separation of resources. If there is also a restriction in terms of which programs (or classes of programs) are allowed access to resources, it is possible to limit the damage that can result from self propagating programs such as computer viruses. By separating resources, the principle of least privilege can be applied to limit the damage a virus could cause if one did get into the system. Such viruses normally act as Trojan horses: while performing operations on some authorized resource, they also covertly perform operations on resources (executable program files) which they should not be permitted to modify. If a program whose purpose is to display mail files, for example, is prevented from accessing other program files because they are a type of resource not appropriate or necessary for the mail reader program to access, the mail reader program is prevented from covertly modifying the other program files while it is performing its normal, authorized function. This topic will be discussed further under encapsulation mechanisms. The remainder of this chapter discusses separation of resources mechanisms grouped under three policies: address space separation, encapsulation, and access control.

4.3.1 Policy of Address Separation

Throughout the history of computer architectures a variety of file and memory protection schemes have been developed to prevent a process from incorrectly accessing a resource or overwriting memory that is not allocated to the process. Overwrite prevention is necessary since such memory often contains another process's code or data, or a portion of the operating system, control tables, or device registers for the system. We describe two address separation mechanisms that can be used to prevent unauthorized access or overwrites: separation of name spaces and descriptors.

4.3.1.1 Mechanism of Separation of Name Spaces

Fundamental to the separation of name spaces is the principle that an object that cannot be named also cannot be addressed. This mechanism, separate from, although often confused with memory protection, is both an elementary way to think about protection and a basis for file protection. In this approach, users are provided distinct separate naming or ``catalog'' spaces, such that the same name (e.g., home directory) refers to separate, non-overlapping objects when used by each user. The result is that each user's processes are unable to name or access the other's named objects, the object's addresses are resolved to a different locational meaning. This was the primary file system protection scheme of M.I.T's Compatible Time Sharing System [Saltzer 1977]. This mechanism can be applied to protecting arbitrary named resources where each user or process has its own distinct name space in which it may use resource names without having to consider name usage by other processes. The principle drawback of this mechanism is that it precludes sharing, at least via named objects. It does not provide for ``controlled sharing'' at all. So other mechanisms have been developed to address protection (e.g., access control lists) and sharing of information (e.g., working directories).

4.3.1.2 Mechanism of Descriptors

The approach to memory protection that is the basis of ``segmented'' memory systems is based on special hardware implementing a descriptor table. In the descriptor based approach, all memory references by a processor are mapped through this special piece of hardware. The descriptor table controls exactly which parts of memory are accessible.

The entries in the descriptor table are called descriptors and describe the location and size of memory segments. A descriptor contains two components: a ``base'' value and a ``bound'' value. The base is the lowest numbered memory address a program may use, and the bound is the number of memory locations beyond the base that may be used. A program controlling the processor has full access to everything in the base bound ranges located in the descriptor table.

The descriptor based approach to memory protection operates on the assumption that the address (name) of an object in memory is equivalent to the object itself, and thus that controlling the ability to use the address to reference the object provides control over the object. The advantages and disadvantages of descriptors are a direct result of this assumption. Because objects in primary memory are always referenced via an addressing mechanism, into which the descriptor based protection mechanism is interposed, it is straightforward to show that this protection mechanism serves as a reference monitor for objects that are in memory.

But not all objects reside in memory; they also reside on other devices, some of which (such as data on conventional 9-track tapes) are not referenced by addresses at the hardware level. Thus, it is necessary to impose control on access to the device, then implement additional protection mechanisms in software to control access to the data stored on the device. This approach complicates verification and is less consistent with the requirement that the reference monitor be ``small enough to be subject to analysis and tests.'' Furthermore, some memory locations, such as the CPU registers, are not protected in this way at all; the approach taken is to view these memory locations as being part of the subject, and thus the protection mechanism seeks to prevent data from being moved into these memory locations.

Granularity of segments is another problem in such systems. Since segment sizes vary, locating available space in memory is not trivial. As a result, fixed sized segments (commonly called pages) are used in most modern computer architectures. Fixed sized segments, pages, also eliminate the need for the ``bound'' component of the descriptor since all pages have the same bound.

One other disadvantage of descriptors is that the attributes of objects protected by descriptor based mechanisms are not attached to the objects in a consistent and direct way. Objects of similar attributes are grouped together where they can be protected under the same descriptor; if values are copied out of them into registers or into other objects, the attributes of the original object do not automatically follow the information. This is one of the principal reasons for the information flow restrictions that result from the traditional mandatory access control disclosure rules: the rules are intended to keep data with similar attributes together. However, there are no theoretical impediments that preclude direct attachment. Rather, this is an implementation issue and reflects the industry's past success and failure.

There are also advantages to the descriptor based approach. It is usually very efficient to implement in hardware, and access rights can be associated with each segment via the descriptors. The block sizes which descriptors reference (the granularity) are chosen specifically for this reason. These block sizes enable the otherwise time consuming arithmetic for computing permissions to be performed by partitioning the binary addresses into groups of digits, and routing these groups of digits through the hardware which processes the addresses to determine access permissions. Thus, the mechanism tends to be time and hardware efficient.

There is also the advantage of experience. Because this approach has been extensively used for several decades, it is well understood and well developed. It has strongly influenced the current view of computer security, as well as the basic machine architecture most people associate with a computer. Thus, existing concepts fit well into this approach.

4.3.2 Policy of Encapsulation

Encapsulation mechanisms provide important design paradigms for producing systems with improved integrity properties. These mechanisms provide support for the type of system architecture requirements of the TCSEC (for B3 level features and assurance) which state, ``the TCB shall incorporate significant use of layering, abstraction, and data hiding'' [DOD 1985, p 38].

Underlying these design paradigms is the goal that software and hardware components should be structured so that the interface between components is clean and well defined, and that exposed means of input, output, and control, besides those defined in the interface, do not exist. This approach serves to limit complexity of the module interfaces, as well as complexity of the software as a whole, since it restricts the means by which the modules of the system can interact. It also serves to impose an architectural discipline on those implementing the system, when provided as a mechanism rather than simply as a design paradigm, because it prevents the implementers from circumventing the defined interfaces in the course of implementation. Four encapsulation mechanisms are discussed below: abstract data types, strong typing, domains, and actors. Given that certain resources within the system are encapsulated, we describe three well-defined interface mechanisms that can be used to manipulate those encapsulated resources: message passing, data movement primitives, and gates.

4.3.2.1 Mechanism of Abstract Data Types

Abstract data types precisely define the semantics of data and control the operations that may be performed on them. This mechanism defines a type to be a particular set of data values and a particular set of operations to perform on those data. Thus, just as there is usually a set of ``integers'' in a programming language, it is possible to define a ``degree'' type which consists of the degrees of a circle, and operations performed on values of this type. More complex types are also possible, and may be used to represent physical objects (robot arms, mechanical equipment, etc.) for programming purposes. Abstract data typing is a fundamental mechanism of modern high order languages.

Abstract data types have substantially improved the ability to support integrity. Their use addresses two of the three goals of integrity cited earlier. They allow the programmer to define data to be of types more directly derived from the things the data actually represents; thus, there is less of a ``semantic gap'' between what the programmer is trying to represent and what form the data actually takes. This supports the goal of ``maintaining internal and external consistency of data.'' With respect to the other two goals, the mechanism, as it is normally used, does not prevent unauthorized users from modifying the data; however, it can aid in preventing authorized users from modifying the data in inappropriate ways, as long as they correctly use the defined operations for that type.

The existence of support for abstract data types does not in itself completely ensure that subjects cannot modify data in inappropriate ways since many implementations allow programmers to override the data typing mechanisms. The adherence to the defined data types in such implementations is the responsibility of the programmer.

4.3.2.2 Mechanism of Strong Typing

The mechanism of strong typing is simply the strong enforcement of abstract data types. The permitted values and operations of an abstract data type are strongly enforced by the compiler or hardware, and cannot be circumvented. Programmers are prevented from taking convenient and possibly efficient shortcuts in their manipulations of the data, but integrity is enhanced by constraining programmers to manipulate data in well-defined ways. Users are prevented from accidentally misusing the data type, especially when they may not understand the details of a particular data type. Even though an object's representation might be compatible with operations not associated with that data type, such operations will not be permitted.

For example, consider a data type whose value represents the angular position of a gun turret. Suppose that the angular position is represented using the computer's hardware representation for a floating point number. Even though this floating point number is allowed by the hardware to take on a very wide range of values, the use of strong typing could prevent the normal floating point operations from being performed on the number. Instead, a new set of operations would be provided for modifying the number, which constrains the resulting values to meet acceptable conditions.

Thus, the ``+'' operation might be defined to prevent a programmer from incrementing the turret position such that its angle was greater than 360 degrees. This ``+'' operation might also prevent a programmer from adding more than a certain increment to the position in a single operation (to meet physical restrictions of the machinery that rotated the turret). This ``+'' operation might also be defined to have the side effect of repositioning the turret to the new value whenever the number was changed, so that the number always accurately reflected the position of the turret. Strong typing would prevent the use the machine's general floating point ``+'' operation instead of the operation defined for the turret-position data type. Thus, it would not be possible to either (1) set the turret position to invalid values, or (2) set the number so it did not accurately reflect the turret position. This form of strong typing is currently available in the Ada programming language and in other high order languages.

Strong typing is not limited to programming languages. For example, the mechanism may be used to prevent a pilot of a particular type of aircraft from commanding it to perform an operation for which it was not designed and that would be damaging to the aircraft's structure. The values to which aircraft controls may be set can be considered analogous to the set of values that data of a particular type may take on. The specific set of allowed operations the pilot may perform on the controls may be considered to be analogous to the allowed set of operations on a conventional data type. Ultimately, the control values and operations may actually be represented via conventional programming language data types in, for example, a ``fly by wire'' system; in such a case, maintaining a consistent mapping between the programming language type restrictions and the user interface (aircraft controls) in a way acceptable to the pilot may prove a significant design challenge.

4.3.2.3 Mechanism of Domains

A domain is a mechanism that uses the principle of separation to protect resources. A domain achieves protection by encapsulating its resources in what is effectively a distinct address space. As described in [Boebert 1990] there are at least two ways to implement domains: as common subsets of subjects, and as equivalence classes of subjects.

Domains are typically implemented as common subsets of subjects. The Multics ring mechanism is a prototypical example of domains implemented in this way. The common subsets are the rings. The address space of each subject is subdivided into a series of rings. The rings have a hierarchical relationship, with the innermost ring being the one with highest privilege. Program code executing in a given ring can access data objects in rings of lower privilege, and they can make procedure calls to rings of higher privilege. Due to the hierarchical relationship among domains in this type of implementation, domains of higher privilege are protected from domains of lower privilege. Protection between mutually suspicious domains cannot be supported.

The LOCK architecture [Boebert 1990] overcomes this problem by implementing domains as equivalence classes of subjects. Each instance of a domain is encapsulated in a single subject, and has its own separate address map to achieve isolation. Shared objects, which are mapped into the address space of two or more subjects, provide the means of data flow between domains. Control flow requires a special mechanism because each subject has its own execution point which stays within the address space of that subject. The mechanism used in LOCK is intersubject signalling.

4.3.2.4 Mechanism of Actors

Programs can be designed to function as actors, in which the user of the program issues a request to access an object in a particular way, and the program then acts on the request itself. In such a case, the program interprets the request and then performs the appropriate action, rather than having the action performed upon the object by an outside entity. An example would be an integer object, which would be given the request ``add 12 to yourself.''

Actors are partly software structuring mechanisms; they help to organize software into distinct, autonomous parts. But they also reduce the risk of ``tampering'' with the object by direct modification, since the only way to cause the object to be modified is by requesting the program (actor) to do so itself. The program can be designed to check requests before acting upon them, and to reject improper requests. Actors are distinct from mechanisms in which the user process ``executes'' a segment of code directly in order to manipulate an object (i.e., a subroutine written to manipulate data of a particular type). The underlying implementation distinctions between the two, however, can be subtle.

The actor concept explicitly requires that individual modules are responsible for performing operations on their data themselves, in response to requests from modules that use them. This is, in part, simply a way of looking at the software. Thus, the same operation ``a+b'' can be thought of as meaning either ``fetch values a and b and perform the `+' operation on them, giving a new value,'' or ``request a to add b to itself and return the new value to the requester.'' The conceptual difference is that the ``fetch and perform'' concept implies that the user directly accesses the values, whereas the latter (actor) concept implies that the values are autonomous entities that respond to user requests.

The actor concept suggests that the values are less passive and thus more able to be self-protecting-a view that can promote a better software architecture from the security standpoint. It is, however, necessary to continue to keep efficiency and the architecture of the underlying hardware in mind when using actors. The higher level of abstraction should not hide the importance of efficient architectural design from the implementer, particularly in embedded systems that often have real-time requirements.

4.3.2.5 Mechanism of Message Passing

Message-passing mechanisms, when used as the primary interface between modules, restricts improper access because requests and data are communicated to the software components through a well-defined mechanism that is distinct from the mechanisms used to modify data directly (e.g., direct writes to memory, input/output operations, branches into module code entry points). Trusted Mach [Branstad 1989] is an example of the use of such a mechanism. Message passing mechanisms, of course, use some of these direct-access mechanisms internally. But the message passing paradigm controls the accesses into components implementing the message passing mechanism, rather than distributing the use of the direct-access mechanisms throughout the system.

As a result, implementation of a reference monitor, as well as verification of the mechanisms used to affect intermodule communication, are simplified by this localization of function. The operations that are directly available to the users (the individual software modules wishing to communicate with one another) do not provide a way to directly modify or execute portions of other modules; they only provide the ability to enqueue well-defined requests to pass data to the other modules. These requests can then be checked for correctness, and the modifications can be carried out at a time when the receiving module is ready to handle the data (i.e., when the receiving module executes a receive message operation).

4.3.2.6 Mechanism of the Data Movement Primitives

Most computer architectures equate a data object with the place in which it is stored. A computer architecture which uses the data movement primitives ``get'' and ``put'' distinguishes a data object from its storage location [Roskos 1984]; it is possible for a process to move (get) an object out of its storage location (which may be shared among many processes) into a private location, perform operations on the object, and then move (put) it back into the shared location. During the time the object is in the private location, it cannot be accessed by other processes, and references to the shared location generate an exception condition. This architecture addresses integrity problems in which multiple processes share data and try to update it at the same time, causing the data object to be corrupted.

4.3.2.7 Mechanism of Gates

When an object (or simply a subroutine) is able to be executed directly by a process, it is possible to limit the actions that the process can perform through the use of gates. A gate is a controlled entry point into a segment of code. It is not possible to enter the code except through the gate, and restrictions can be enforced regarding which processes are allowed to enter the gate (e.g., only code with certain privileges, or acting on behalf of a given user, may be allowed to do so).

Gates can also be used to cause a transition in privilege. For example, a process which enters a routine via a gate may gain extra access permissions or allowed operations, since use of the gate ensures that a specific segment of code will be executed while these permissions are granted. The code can be verified to ensure that it does not allow unauthorized use of the privileges. Upon exit from the routine, the extra privileges are revoked by the system.

4.3.3 Policy of Access Control

The policy of access control restricts which data a subject is allowed to access. An evolving framework for addressing restrictions on access control is given in [Abrams 1990]. Restrictions can be either identity-based or rule-based. Identity-based controls are typically associated with discretionary security policies, and rule-based controls are typically associated with mandatory security policies. In identity-based mechanisms, the identity of a subject or object must match the identity specified in the particular access control mechanism. We discuss three mechanisms that can be used to enforce identity-based access restrictions: capabilities, access control lists, and access control triples. In rule-based mechanisms, one or more attributes of a subject must obey specified rules in relation to corresponding attributes of an object that the subject desires to access. We discuss labels as a mechanism to enforce rule-based access restrictions.

4.3.3.1 Mechanism of Capabilities

Capabilities provide a general and flexible approach to access control. A capability is a specially protected object (or value representing it) that contains the name of the object to which it refers along with a specific set of attributes or permissions that specify what types of access the possessor of the capability is granted. It can be thought of as an unforgeable ticket, which, when presented, can be taken as incontestable proof that the presenter is authorized to have the specified access to the object named in the ticket. Thus, when a subject wants to access an object, the subject simply presents the appropriate capability and a descriptor is immediately created (giving the subject access to the object) with no additional permission checks being performed.

The system can invalidate a capability (so that possession of it no longer grants access), can control whether or not it is possible to give away the capability to another process, or can limit the number of times a capability may be used to access an object. Capabilities provide a very powerful and flexible means of controlling access to resources, and distributing that access to others. This topic is discussed further and many implementations are identified by Gligor et al. [1987].

4.3.3.2 Mechanism of Access Control Lists

Where capabilities employ a ticket-oriented strategy to access control, access control lists (ACLs) employ a list-oriented strategy. An access control list contains the names of subjects that are authorized to access the object to which it refers, as well as specific permissions that are granted to each authorized subject. Thus, when a subject wants to access an object, the system searches for an entry for the subject in the appropriate ACL. If an entry exists, and if the necessary permissions are part of that entry, then a descriptor is created. The list-oriented strategy requires one more permission check than the ticket-oriented approach when a subject requests access to an object.

ACLs can be thought of as a somewhat more flexible form of strong typing, although conceptually they are not as powerful for programming purposes. Their generality allows one to set access permissions arbitrarily, with the result that someone who does not fully understand an application might grant a user access to programs and data which were incompatible. This could result in a program operating upon data it was not designed to handle, and thus performing incorrectly. This situation is largely an educational concern for users, but is of importance in the design of programmable systems.

In a system in which names are considered equivalent to objects, such that controlling the use of names controls access to the objects they reference, a potential problem exists if multiple names are allowed to reference the same object. These names essentially provide alternate ``paths'' to the same object, and if ACLs are associated with the names rather than with the object itself, a user's access may be restricted if a reference is made via one name, but not restricted if the reference is made via another name. The result is that a given user's access permissions for an object can be inconsistent when viewed in terms of these multiple names. Multiple names can be a problem in database systems where multiple ``views'' are provided with access controls associated with the views. It is also a problem in conventional operating systems when access permissions are associated with directories and multiple ``links'' (directory references) exist to the same file. This problem exists regardless of whether the access permissions for files are stored in the directories, or whether access controls on the directories themselves are used to control access to the objects referenced within the directories.

4.3.3.3 Mechanism of Access Control Triples

An access control triple is a list-oriented mechanism that was conceptually introduced by Clark and Wilson [1987] to address access control requirements more commonly associated with ``commercial'' computer systems rather than ``military'' systems. It provides a finer granularity of access control than normally associated with capabilities and ACLs by specifying required linkage for a user, program, and object along with specific permissions that are granted to the user while invoking the particular program on the given object. The triples of the Clark-Wilson model essentially define allowed type operations, and the Integrity Verification Procedures (IVPs) of the Clark-Wilson model define a set of permitted values for data of a given type.

A principal distinction, then, is the association of a given set of triples with a given object, rather than with a given type. This distinction is necessary because the Clark-Wilson triples control which users may perform operations on a given object, and thus provide one of the components of integrity enforcement which is missing from conventional strong typing. To do this form of integrity enforcement, access permissions must be specified on a per object basis, not simply by type. ACLs also specify these access permissions, but they do not include a detailed specification of what operations may be performed: they simply indicate whether or not each predefined set of operations is permitted.

Access control triples are not a strict generalization of strong typing because the per object specification property of Clark-Wilson does not allow triples to implement conventional strong typing. A single triple describes an existing object, not a set of objects that will be created in the future. A similar argument can be made with regard to ACLs. That is, ACLs permit a wide variety of operations to be performed without explicitly specifying them as long as they do not violate the access restrictions imposed by the ACLs.

4.3.3.4 Mechanism of Labels

It is also possible to associate one or more labels with each user, process, and resource to implement rule-based access control. Processes inherit user labels, and a process is granted access to a resource only if their two labels meet some criterion of comparison. For example, a TCSEC mandatory access control label has both a hierarchical and a non-hierarchical component. Each subject is assigned a clearance and one or more categories or compartments, and each object is typically assigned a classification and one category. A subject is granted access to an object based on a comparison of these values. In this case, the comparison is via an ordering relation rather than an equivalence relation, but the underlying principle is the same. This mechanism also addresses the concepts of rings and privilege states, in which a process is granted access to successively greater numbers of resources as its label increases in value according to some ordering relation.

The underlying concept of labels has analogies to capabilities in the sense that the subject has ``possession'' of a particular value which gives the subject access to certain objects. As with capabilities, the subject cannot change the value stored in the label; unlike capabilities, the subject can only have one access label value per rule set at a given time. For example, a user may be cleared to the Secret level, but if he has logged on at the Unclassified level, the rule set will not allow him to access data at the Secret level.

A label is a relatively simple mechanism that can serve as the basis for a number of more complex mechanisms. The principal limitation is that a subject can only have one access label value per rule set at a given time. This fact limits the flexibility of the mechanism. Increasing the number of labels per subject or object limits their benefit, since complexity of label assignment rapidly increases unless a structured mechanism such as capabilities, access control lists, or access control triples is used instead.

4.4 POLICY OF FAULT TOLERANCE

Systems that support mission-critical applications must not only preserve integrity to the extent possible, but also must detect and attempt to correct integrity failures that result from unavoidable situations, such as physical failures of equipment or media. Systems which do this support a policy of fault tolerance.

The policy of fault tolerance potentially involves two parts. The first part is the detection of errors. Detection is necessary for a system to determine when a failure has occurred. Detecting errors that are corrected by the system is as essential as detecting failures that cannot be corrected, since such failures may indicate a potential degradation of system performance, or may indicate that the system is approaching a threshold at which errors are no longer correctable. We will discuss fault tolerant error detection mechanisms under the policy of summary integrity checks. The second part of the policy of fault tolerance is the attempted correction of errors. We will discuss fault tolerant error correction mechanisms under the policy of error correction.

4.4.1 Policy of Summary Integrity Checks

All mechanisms supporting the policy of summary integrity checks provide a ``second copy'' of a group of data against which the first copy may be compared in order to detect changes. Nearly all of these mechanisms operate by producing a smaller piece of data which is computed from the collection of data whose integrity is to be verified. This approach has an inherent weakness because reducing the size of the summary data will result in loss of information, except to the extent that the original data has redundancies which can be removed by data compression techniques. The result is there will be more than one value of the original data for which the summary data is identical, and by judiciously changing the original data, the computed summary data may still indicate that the data is unchanged. This ``judicious changing'' type of attack is used by some virus programs to circumvent checksum programs intended to detect the viruses. We describe five summary integrity check mechanisms below: transmittal lists, checksums, cryptographic checksums, chained checksums, and the check digit.

4.4.1.1 Mechanism of Transmittal Lists

When batches of data are stored or transmitted together, a transmittal list may be attached to the data. This list identifies what data are included in the batch, and may be used to verify that no data are missing. Transmittal lists are used to provide confirmation that all the components of a multipart data object are present, but do not generally provide for detection of modification of the components still present, unless combined with other techniques. The same is true with data counts, a generalization of a transmittal list in which simple counts of data are used to ensure that no data are missing. Data counts are weaker than transmittal lists in the sense that they do not provide information on missing components.

4.4.1.2 Mechanism of Checksums

When a block of data is transferred from one point to another, a checksum is often added to the block of data to help detect errors. A checksum is a numeric value that is computed based on the entire contents of the data. When the original data is created, a checksum is calculated and appended to it. The checksum is then regenerated when the data is received at its destination or, in some applications, when the data is read from memory. The regenerated checksum and the original checksum are then compared to determine if an error has occurred. Checksums do not, however, protect the data from destruction; they only provide a means for detecting when the data has been tampered with.

A checksum is one of the stronger mechanisms for integrity checking. It can be very difficult for an adversary to change the data without causing the checksum computed from the changed data to be different from the checksum computed from the original data. There are two properties of checksums that must be considered. First, checksums need to be accompanied by other mechanisms (e.g., time stamps) to prevent replay. Second, it must be realized that if the check- sum has fewer bits (contains less information) than the data itself, there will always be more than one value of the data that will result in the same checksum value. Given this fact, the function used to compute the checksum should be such that changing a single, small portion of the data will result in a high probability that the checksum will change. Also, the set of all possible values of the data should produce all possible values of the checksum, with the probability of generating a given checksum being evenly distributed for all possible values of the data.

4.4.1.3 Mechanism of Cryptographic Checksums

When data is transmitted over an open communications medium, where both the data and the checksum are exposed to modification, the original data may be vulnerable to undetected modification. An adversary may be able to modify the data, recalculate the checksum, and store them in place of the original pair. To counteract this threat, cryptographic techniques are used. Once a checksum is computed, it is encrypted with a protected encryption ``key.'' If the checksum is encrypted, an adversary must gain access to the encryption algorithm and key to change the checksum to indicate that the data is still unmodified. With protected keys, the encryption algorithm may be assumed to be known or discoverable by the adversary without harm as long as the specific key remains unknown. Keeping both the algorithm and the keys protected increases the work factor of the adversary in attacking the system, but may also increase the cost of the system.

4.4.1.4 Mechanism of Chained Checksums

Checksums (either conventional or cryptographic), when applied to sequenced data, may be made a function of the previously seen data. In accounting, these chained check- sums are termed run-to-run totals. For example, a checksum may be computed across multiple data, each of which has a checksum itself. This method is used to give assurance via a single number that all preceding data is unaltered. However, as the amount of data included in the chain or sequence increases, the probability of undetected errors increases because there are more data values that will give the same checksum as the number of bits of data increases in comparison to the number of bits of the checksum. Thus, chained checksums have the advantage that they can detect deletion of portions of the data, but have the disadvantage that the size of the data being checked can become large in proportion to the size of the checksum.

4.4.1.5 Mechanism of the Check Digit

A check digit is simply a special case of a checksum, where the checksum is a single digit. It is normally used only to check for gross unintentional errors, such as mistyping of a numeric identifier code. A check digit does not provide much protection against malicious attacks or against noisy transmission lines because the probability is relatively high that even a randomly chosen digit will occasionally be correct. A ``parity bit'' is an example of a binary check digit.

4.4.2 Policy of Error Correction

In addition to simply detecting incorrect data, it is possible to use methods to correct errors in the data. The simplest approach to error detection would be to provide a certain number of redundant copies of the data, possibly by different channels, and then compare these at the time when it is desired to determine whether the data integrity has been violated. This concept can be extended to error correction if it is possible to tell which of the redundant copies of the data has not been altered. Various error correction methods give varying probabilities of retrieving the original, unaltered data. We describe two of these redundant copy methods below: duplication protocols and handshaking protocols.

The process of providing redundant data can be improved upon in terms of efficiency if assumptions are made about the types of errors likely to occur. In particular, if it is expected that only a small portion of any one block of data is likely to be altered, it is possible to reduce the amount of redundant data that has to be sent in proportion to the number of bits of data expected to be changed. Error correcting codes can be used to achieve error correction with less than a completely redundant copy of the original data.

4.4.2.1 Mechanism of Duplication Protocols

In environments in which there is only one-way data communication, such as data distributed by one-way radio broadcast, duplication protocols are used. These generally involve duplicating the data being sent. One approach involves duplication in time, re-sending the same data value several times, along with check information. For example, a standard data protocol used in commercial shipping communications involves representing each distinct data value as a binary number, the sum of whose digits is a constant number, and sending small groups of consecutive values several times. If it is found that one of the data values consists of digits that do not add up to the constant, the value is assumed incorrect, and one of the redundant copies of the data is tried. By timing the resending of redundant information according to known error-generating properties of the medium, it is possible to minimize the probability that all copies of the data value will be in error. The sending of small groups of values several times rather than immediately repeating each value before proceeding to the next serves to increase the likelihood that an error-generating phenomenon will have abated before the data are resent.

Another approach to duplication of one-way data is by sending it by multiple media. In the case of radio transmissions, the data may be sent on several frequencies at the same time, since phenomena that interfere with communication on one frequency often do not affect a different frequency at the same time. In the case of communications by networks, the data can be sent simultaneously by several paths. These approaches are essentially a form of redundant processing.

The simple redundancy exemplified by duplication protocols has a principal benefit of simplicity; the algorithms needed to regenerate the data are simpler than those of more compact error correction codes, and the density of error allowed may be greater. Use of a diversity of media likewise allows for a greater density of errors; one or more of the redundant channels may fail completely as long as one other channel continues to function without error. But considerably more hardware resources may be required for such an approach to duplicate all the components that may fail.

4.4.2.2 Mechanism of Handshaking Protocols

Handshaking protocols are one of the most frequently seen approaches to error correction where two-way communication exists, as in computer networks. In handshaking protocols, a block of data is sent, along with a check value for use with a summary integrity check mechanism; the check value is used to check the transmitted block for error on the receiving end. A reply is then returned to the sender indicating whether or not the received data checked as valid. If an error is detected, the receiver repeatedly requests the data until it is received without error. An example of a handshaking protocol is the X.25 protocol used in hardwired and radio-based communications networks.

This protocol is complicated by the fact that either messages or replies may be entirely lost, and thus such a protocol must handle non-responses as well as negative responses. This is the reason that messages indicating successful transmission are returned, not just unsuccessful transmission. Furthermore, efficient use of the communications medium may require that long delays not be incurred waiting for a reply to come back from the receiver; in such a case, later blocks of data may be sent while awaiting a reply to the first block. If a reply eventually comes back indicating that a given block was not received, it can be resent out of sequence. This in turn requires use of the chained checksum mechanism discussed earlier, to ensure that all blocks are received and are put in the proper order.

To avoid building up an excessive backlog of unacknowledged messages, there is usually a ``window'' of unacknowledged messages that are allowed to be outstanding; if more messages would be sent than are allowed by this limit, sending stops until the backlog is filled. Some protocols which allow sending messages in two directions at the same time obtain further efficiency by combining acknowledgement messages for data messages going in one direction with data messages going in the opposite direction, and vice versa. The design of these protocols can become quite complex, but the area is well researched and can result in very efficient protocols. On the other hand, an isolated error in a message may require the entire message to be resent, and thus the size of the message unit in proportion to the frequency of errors is an important consideration in designing these protocols. This mechanism is commonly used in supporting the real-time requirements of most computer systems, although it is omitted from certain high-performance computer systems.

4.4.2.3 Mechanism of Error Correcting Codes

Duplication and handshaking protocols assume that failure occurs at an intermediate point beyond the sender of the data. If the sender's copy of the data is damaged, these protocols will not be able to recover the data. Thus, an error correcting code must be used on the original copy. For example, Hamming codes employ the concept of overlapping parity to achieve error correction of original data.

The Hamming single error correcting code is formed by partitioning the original data bits into parity groups and specifying a parity bit for each group. The ability to locate which bit is erroneous is obtained by overlapping the groups of bits. A given data bit will appear in more than one group in such a way that if the bit is erroneous, the parity bits that are in error will identify the erroneous bit. For example, suppose that there are four data bits (d3,d2,d1,d0) and, as a result, three parity check bits (c1,c2,c3). The bits are partitioned into groups as (d3,d1,d0,c1), (d3,d2,d0,c2), and (d3,d2,d1,c3). Each check bit is specified to set the parity, either even or odd, of its respective group. Now, if bit d0, for example, is erroneous, both c1 and c2 are incorrect. However, c3 is correct because the value of d0 has no impact on the value of c3. Therefore, the error in bit d0 can be corrected.

Typically, an error correcting code is used to allow some limited portion of the original message to be regenerated, given the correct value of the remaining portion. As the amount of lost information that the code is supposed to be capable of regenerating increases, the size of this code increases, so a tradeoff is made based on the expected rate of error in the data. If the error rate is low, a smaller code may be used, since the probability of a large number of errors in a single block decreases.

5 INTEGRITY MODELS AND MODEL IMPLEMENTATIONS

A computer security model is a high-level specification or an abstract machine description of what a system does [Goguen 1982]; it provides a framework for linking a security policy (security requirements for a given system) to the mechanisms that are used to implement the policy. While models describe what types of mechanisms are necessary to satisfy an integrity policy, model implementations describe how specific mechanisms can be used together in a system to achieve integrity protection required by a particular security policy. Several models and model implementations are examined in this chapter.

5.1 INTEGRITY MODELS

In this section, we describe and analyze five models that suggest different approaches to achieving computer integrity: Biba, Goguen and Meseguer, Sutherland, Clark and Wilson, and Brewer and Nash. (footnote #1) Three of these models (Goguen and Meseguer's, Sutherland's, and Brewer and Nash's) were not originally intended as integrity models, but we show how they can be viewed as applying to the needs of integrity. Although all five of these approaches establish sound restrictions for limiting the manipulation of data, we have not been able to empirically determine whether any of the models are complete or sufficient with respect to integrity.

[1] The Clark and Wilson model is not stated in formal mathematical terms like the others, but formal axioms are not necessary to compare the advantages and disadvantages of each model.

5.1.1 Biba Model

5.1.1.1 Discussion of Biba

The model defined in [Biba 1977] was the first of its kind to address the issue of integrity in computer systems. This approach is based on a hierarchical lattice of integrity levels. The following basic elements are used to mathematically define the Biba model:

S: the set of subjects s; the active, information processing 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;

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;

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

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

m: a relation (subset of S x O) defining the capability of a subject, s element_of S, to modify an object, o element_of O: s m o; i:a relation (subset of S x S) defining the capability of a subject, s1 element_of S, to invoke another subject, s2 element_of S: s1 i s2.

Integrity is evaluated at the subsystem level. A subsystem is some subset of a system's subjects and objects isolated on the basis of function or privilege; a computer system is defined to be composed of any number of subsystems, including only one. Integrity threats are classified by Biba as being either internal or external to a subsystem. An internal threat arises if a component of the subsystem is malicious or incorrect. An external threat is posed by one subsystem attempting to change (improperly) the behavior of another by supplying false data or improperly invoking functions. Biba feels internal threats are sufficiently handled by program testing and verification techniques; the Biba model then only addresses external threats.

The Biba model elements support five different integrity policies, which are described below: the Low-Water Mark Policy, the Low-Water Mark Policy for Objects, the Low-Water Mark Integrity Audit Policy, the Ring Policy, and the Strict Integrity Policy. There is a sixth policy, called the Discretionary Integrity Policy, based on ACLs and the Ring Policy that actually describes a specific proposal for modelling integrity protection in the Multics operating system. Since ACLs are discussed earlier and the Ring Policy is discussed here, the Discretionary Integrity Policy is not addressed separately.

5.1.1.1.1 Low-Water Mark Policy

In this policy, the integrity level of a subject is not static, but is a function of its previous behavior. The policy provides for a dynamic, monotonic, 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, and is formalized by the following axioms:

for_all s element_of S, o element_of O

s m o ==> il(o) leq il(s)

for_all s1,s2 element_of S

s1 i s2 ==> il(s2) leq il(s1)

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

il'(s) = min {il(s), il(o)}

where il'(s) is the integrity level of s immediately following the access.

5.1.1.1.2 Low-Water Mark Policy for Objects

In addition to changing the integrity level of subjects, the Low-Water Mark Policy for Objects postulates that the integrity level of modified objects also changes. This alternate policy can be characterized by the following rules.

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

il'(s) = min {il(s), il(o)}

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

il'(o) = min {il(s), il(o)}

5.1.1.1.3 Low-Water Mark Integrity Audit Policy

This audit policy is an unenforced variant of the Low-Water Mark Policy for Objects. It provides a measure of possible corruption of data with ``lower'' integrity level information. A ``current corruption level'' (cl) for subjects and objects is defined in the following manner.

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

cl'(s) = min {cl(s), cl(o)}

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

cl'(o) = min {cl(s), cl(o)}

The value of cl for an object represents the least integrity level of information which could have been used to modify the object.

5.1.1.1.4 Ring Policy

The Ring Policy provides kernel enforcement 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. Flexibility of the system is substantially increased by allowing observations of objects at any integrity level. The policy is defined by two axioms:

for_all s element_of S, o element_of O

s m o ==> il(o) leq il(s)

for_all s1,s2 element_of S

s1 i s2 ==> il(s2) leq il(s1)

5.1.1.1.5 Strict Integrity Policy

The Strict Integrity Policy is the mathematical dual of the confidentiality policy presented in the TCSEC [DOD 1985]. It consists of three parts: a Simple Integrity Condition, an Integrity *-property, and an Invocation Property. The Simple Integrity Condition states that a subject cannot observe objects of lesser integrity. Written in Biba's mathematical notation

for_all s element_of S, o element_of O

s o o ==> il(s) leq il(o)

This rule constrains the use of objects (data or procedures) to those whose non-malicious 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). Biba considers execute access to be equivalent to observe access, so objects must have an integrity level greater than or equal to that of the requesting subject in order to be executed.

The Integrity *-property states that a subject cannot modify objects of higher integrity. Written in Biba's notation

for_all s element_of S, o element_of O

s m o ==> il(o) leq il(s)

This rule ensures that objects may not be directly modified by subjects possessing insufficient privilege. The rule, however, assumes that the modifications made by an authorized subject are all at the explicit direction of a non-malicious program. The unrestricted use of subsystems written by arbitrary users (to whose non-malicious character the user cannot attest) does not satisfy this assumption; the Simple Integrity Condition guarantees this assumption is satisfied.

The Invocation Property states that a subject may not send messages to subjects of higher integrity. Mathematically, the integrity level of the receiving subject must be less than or equal to the integrity level of the sending subject:

for_all s1,s2 element_of S

s1 i s2 ==> il(s2) leq il(s1)

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. Therefore, this rule follows directly from the Integrity *-property.

5.1.1.2 Analysis of Biba

Biba defines integrity as a relative measure; it is not absolute. According to this definition, a subsystem possesses 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 to determining whether or not the subsystem possesses integrity; all that is required is that the subsystem adhere to the code of behavior. For this model, the goal of computer system integrity is thus the guarantee that a subsystem will perform as it was intended to perform by its creator.

This is a rather broad interpretation of integrity. The goal of every computer system should be to perform as it was intended to perform; the real issue is whether the creator designed the system in a manner that can achieve integrity. The Biba model provides a hierarchical lattice for identifying authorized users and providing separation at the user type level. These attributes allow the Biba model to address the first goal of integrity identified at the beginning of the paper: preventing unauthorized users from making modifications.

Of the integrity policies discussed by Biba, the Strict Integrity Policy is by far the most widely addressed, so much so that this policy is often assumed when the Biba model is discussed. The Strict Integrity Policy is the dual of one of the most common and most thoroughly studied computer security policies/models, Bell and LaPadula.

A drawback to Biba's Strict Integrity Policy is how to assign appropriate integrity labels. The Bell and LaPadula model fits the government classification system (e.g., hierarchical levels like Top Secret and Secret, and non-hierarchical categories like NATO). There are established criteria for determining which disclosure levels and categories should be given to both personnel (subjects) and documents (objects). There are currently no corresponding criteria for determining integrity levels and categories. In fact, the use of hierarchical integrity levels, which suggests that some determination will be made about the ``quality'' of data and the ``quality'' of users, seems inapplicable to most practical problems.

5.1.2 GOGUEN AND MESEGUER MODEL

5.1.2.1 Discussion of Goguen and Meseguer

Goguen and Meseguer [1982] introduce an approach to secure systems that is based on automaton theory and domain separation. Their approach is divided into four stages: first, determining the security needs of a given community; second, expressing those needs as a formal security policy; third, modelling the system which that community is (or will be) using; and last, verifying that this model satisfies the policy. Their paper focuses on stages two (policy) and three (model); this discussion focuses on the model.

Goguen and Meseguer distinguish sharply between a security policy and a security model. A security policy is defined as the security requirements for a given system (based on the needs of the community). In general, security policies are very simple, and should be easy to state in an appropriate formalism. Goguen [1982, p 11] provides a very simple requirement language for stating security policies, based on the concept of noninterference, where

> one group of users, using a certain set of commands, is noninterfering with another group of users if what the first group does with those commands has no effect on what the second group of users can see.

A security model is defined as an abstraction of the system itself; it provides a basis for determining whether or not a system is secure, and if not, for detecting its flaws. A high-level specification or an abstract machine description of what the system does are examples of a security model. The authors develop a set theoretic model which is a sort of generalized automaton, called a ``capability system.'' They achieve noninterference by separating users into different domains; a domain is defined as the set of objects that a user has the ability to access [NCSC 1988]. The Goguen and Meseguer model has an ordinary state machine component, along with a capability machine component which keeps track of what actions are permitted to what users.

5.1.2.1.1 Ordinary State Machine Component

The model is introduced by starting with the classic case in which what users are permitted to do does not change over time. It is assumed that all the information about what users are permitted to do is encoded in a single abstract ``capability table.'' The system will also have information which is not concerned with what is permitted; this information will include users' programs, data, messages, etc. In the Goguen and Meseguer model, a complete characterization of all such information is called a state of the system, and S is defined as the set of all such states. The system will provide commands that change these states; their effect can be described by a function do: S x U x C -> S

where C is the set of state changing commands and U is the set of users. It might be that if user u is not permitted to perform a command c, then do(s,u,c) = s; such security restrictions can be implemented by consulting the capability table, and are simply assumed to be built into the function do.

It is also assumed that for a given state and user, we know what output (if any) is sent to that user. This aspect of the system can be described by a function

out: S x U -> Out

where Out is the set of all possible outputs (e.g., screen display states, listings, etc.). It is assumed that all information given to users by the system is encoded in this function. Putting this all together and adding an initial state, the result is an ordinary state machine M consisting of the following:

a. A set U whose elements are called ``users.''

b. A set S whose elements are called ``states.''

c. A set C whose elements are called ``state commands.''

d. A set Out whose elements are called ``outputs.''

together with

a. A function out: S x U -> Out which ``tells what a given user sees when the machine is in a given state,'' called the output function.

b. A function do: S x U x C -> S which ``tells how states are updated by commands,'' called the state transition function.

c. A constant s0, the initial machine state, an element of S.

The connection with the standard form of the definition of state machine is to take U x C to be the set of inputs.

5.1.2.1.2 Capability Machine Component

In order to handle the dynamic case, in which what users are permitted to do can change with time, it is assumed that in addition to the state machine features there are also ``capability commands'' that can change the capability table. The effects of such commands can be described by the function cdo: Capt x U x CC -> Capt

where Capt is the set of all possible capability tables, U is the set of all users, and CC is the set of capability commands. If a user u is not allowed to perform the capability command c on the table t, the cdo(t,u,c) may be just t again; as before, this security restriction is determined by consulting the capability table. The capability machine component is itself a state machine, with state set Capt and input set U x CC; it models the way in which the capability table is updated, and includes such possibilities as passing and creating capabilities.

5.1.2.1.3 Capability System

The entire capability system is a cascade connection of the capability machine component with the ordinary state machine component. In Figure 2, illustrating the cascade connection, the function Check returns the information from the capability table needed to determine whether or not a given command is authorized for a given user.

(Figure 2. Not available for electronic version.)

Figure 2. Cascade Connection of Capability System

In order to distinguish capability commands from state commands, the definition of a capability system denotes the set of all state commands by SC. So that the state transitions and the outputs can be checked for security against the capability table, a capability table component is added to the state transition function and to the output function. Adding all this to the definition of an ordinary state machine, and also adding an initial capability table, the result is a capability system M consisting of the following:

a) A set U whose elements are called ``users.''

b) A set S whose elements are called ``states.''

c) A set SC whose elements are called ``state commands.''

d) A set Out whose elements are called ``outputs.''

e) A set Capt whose elements are called ``capability tables.''

f) A set CC whose elements are called ``capability commands.''

together with

a A function out: S x Capt x U -> Out which ``tells what a given user sees when the machine, including its capability component, is in a given state,'' called the output function.

b) A function do: S x Capt x U x SC -> S which ``tells how states are updated by commands,'' called the state transition function.

c) A function cdo: Capt x U x CC -> Capt which ``tells how capability tables are updated,'' called the capability transition function.

d) Constants t0 and s0, the ``initial capability table'' and the ``initial machine state,'' respectively elements of Capt and of S.

5.1.2.2 Analysis of Goguen and Meseguer

Goguen and Meseguer's model is based on automata theory with all of the state transitions predefined. The predefined transitions address the first goal of integrity (preventing unauthorized users from making modifications). A significant advantage of having the model based on a standard notion like the automaton is that extensive literature and well-developed intuition become immediately applicable to the problem domain. The fact that the states and corresponding transitions are predefined is both a blessing and a curse. It is a great advantage to know exactly what the system will do at any particular instant. However, it will be very difficult to define all of the states and all of the transitions from one state to another. It will also be difficult to design the system in a flexible manner so that new states can be easily added to the existing design.

Capabilities (as opposed to descriptors) provide the flexibility necessary to implement domain separation. But traditional capabilities cannot control the propagation, review, and revocation of access privileges [Gligor 1987]. It is unclear whether traditional capabilities can be modified to realistically overcome these weaknesses.

5.1.3 SUTHERLAND MODEL

5.1.3.1 Discussion of Sutherland

Sutherland [1986] presents a model of information that addresses the problem of inference (e.g., covert channels). Sutherland uses a state machine as the basis of his model, but he generalizes the model, apparently to avoid limiting it to the semantic details of one particular type of state machine. Thus, his state machine consists of the following:

a) A set of states

b) A set of possible initial states

c) A state transformation function mapping states to states

For each possible initial state, there is an execution sequence defined for each sequence of possible state transformations starting from that initial state. Sutherland generalizes the state machine's set of execution sequences as a set W of all such execution sequences, which he terms ``possible worlds.'' A given execution sequence or ``possible world'' is denoted w, where wW. The ``possible world'' of Sutherland's model is actually even more abstract than we have represented here. As illustrated in a ``state machine instantiation'' appearing later in [Sutherland 1986], under one interpretation of the model, an element w of the set W may consist not only of states, but of ``signals'' (analogous to the ``requests'' and ``decisions'' of the Bell-LaPadula model) interspersed between the states.

Sutherland formally represents the information obtainable from a subsequence of w by defining a set of information functions, fi. Each fi represents the information that can be obtained from one view of w. For example, assume a user has full access to the subsequence of w needed to compute f1(w). If this user knows of an interdependence (covert channel) between f1 and another information function, f2, to which the user may have been prohibited access, the user can infer at least some information about f2(w) from the user's knowledge of f1(w). Sutherland presents this inference formally as follows.

1. The user knows f1(w)=x.
2. The user deduces w element_of S where S={y | f1(y)=x}.
3. The user deduces f2(w) element_of T where T=f2(S).
4. If there is an interdependence between f1 and f2 such that

   there_exists z element_of f2(W)[z not_element_of T]

   the user deduces

   f2(w) =/= z.

Steps 1-3 above are straightforward generalizations from what the user knows of f1; the f2 in step 3 could be an arbitrary function, and calling it f2 in step 3 only anticipates step 4.

It is in step 4 that the user makes the significant inference. The inference results specifically from the user's knowledge that when the result of a particular subsequence of states is seen, it is impossible for the system to have produced the result z. In such a case, the user is able to conclude that the system has not produced a particular result to which the user may have been denied direct access. The extent to which this inference is useful depends on how much information is represented by knowing that result z was not produced. If the user knows that only two results are possible, knowing ``not z'' would be of considerable value. Since the Sutherland model is so general, it does include this and similar inferences.

A concrete example of such an inference would be the observation ``when program 1 produces result x, due to a flaw in the system, program 2 will be unable to compute result z.'' Or, ``while the weapons control system is under test, the target tracking system will not recognize a new target.'' Sutherland defines the case in which information flows from f1 to f2 as the case in which a user seeing f1(w) is able to infer that f2(w) =/= z.

Sutherland goes on to prove a theorem identifying cases in which inference is possible (specifically, cases in which f1 and f2 are dependent), and from this theorem derives an important corollary: that information flows are always symmetric. This corollary has importance to integrity since it shows that the user who can control the computation of f1 can influence the result of f2. This situation can be thought of as a reverse covert channel.

5.1.3.2 Analysis of Sutherland

Sutherland's corollary that information flows are always symmetric suggests that the model may have applicability to integrity. Sutherland speaks in terms of an observer noting that the value of one information function f1 influences another information function f2's set of possible values at a given point in time. If a user can influence the subsequence w which is the argument to f1, and if information flows from f1 to f2, the user can at least partially influence the value resulting from f2. In terms of our previous example, a user who is able to initiate a test of the weapons control system will be able to cause the target tracking system to malfunction, even though testing may have been thought to be a harmless operation. Thus, the Sutherland model addresses the first goal of integrity: preventing unauthorized users from making modifications.

The type of integrity violation represented in the Sutherland model is analogous to a traditional covert channel. Whereas a covert channel communicates data from within a protection domain that should be inaccessible to an outsider, a violation of the type represented by the Sutherland model allows an outsider to influence the outcome of processing occurring within an otherwise inaccessible protection domain.

The Sutherland model differs from some other integrity models in that it does not emphasize a specific abstract protection mechanism. In the last section of the model's exposition [Sutherland 1986, p 177], Sutherland does define a legal_to_get predicate which represents abstract access control, and the formal definition of security is given as

$$\mathit{information\_flows}(f2,f1) ==> \mathit{legal\_to\_get}(f1,f2).$$

Since f1 can be an abstraction for a subject (i.e., the information ``known'' by the subject or in the subject's memory), this predicate can represent access restrictions placed on a particular subject, as well as information flow restrictions between individual data repositories, such as files or ``objects''. But the Sutherland model is primarily involved with representing the goal of protection rather than with a means of enforcing it.

5.1.4 CLARK AND WILSON MODEL

5.1.4.1 Discussion of Clark and Wilson

Clark and Wilson [Clark 1987, 1989] make a distinction between military security and commercial integrity, and present a model for achieving data integrity. They defend two conclusions. First, security policies related to integrity, rather than disclosure, are of highest priority in the commercial data processing environment. Second, separate mechanisms are required for enforcement of these policies, disjoint from those in the TCSEC [DOD 1985]. This model has sparked the information systems and computer security communities to press forward with integrity-related research.

There are two keys to the Clark and Wilson integrity policy: the well-formed transaction and separation of duty. A well-formed transaction is structured so that a user cannot manipulate data arbitrarily, but only in constrained ways that preserve or ensure the internal consistency of the data. Separation of duty attempts to ensure the external consistency of data objects: the correspondence between a data object and the real world object it represents. This correspondence is ensured indirectly by separating all operations into several subparts and requiring that each subpart be executed by a different person.

The Clark and Wilson model is defined in terms of four elements: constrained data items (CDIs), unconstrained data items (UDIs), integrity verification procedures (IVPs), and transformation procedures (TPs). CDIs are data items within the system to which the integrity model must be applied. UDIs are data items not covered by the integrity policy that may be manipulated arbitrarily, subject only to discretionary controls. New information is fed into the system as a UDI, and may subsequently be transformed into a CDI. IVPs and TPs are two classes of procedures that implement the particular integrity policy. The purpose of an IVP is to confirm that all of the CDIs in the system conform to the integrity specification at the time the IVP is executed. An IVP checks internal data consistency, and also may verify the consistency between CDIs and external reality. TPs correspond to the concept of well-formed transactions discussed above. The purpose of TPs is to change the set of CDIs from one valid state to another.

There are nine certification (C) and enforcement (E) rules that govern the interaction of these model elements. Certification is done by the security officer, system owner, and system custodian with respect to an integrity policy; enforcement is done by the system. The rules are stated as follows:

> C1: All IVPs must properly ensure that all CDIs are in a valid state at the time the IVP is run.
>
> C2: All TPs must be certified to be valid. That is, they must take a CDI to a valid final state, given that it is in a valid state to begin with. For each TP, and each set of CDIs that it may manipulate, the security officer must specify a relation, which defines that execution. A relation is thus of the form: (TPi,(CDIa,CDIb, CDIc, ...)), where the list of CDIs defines a particular set of arguments for which the TP has been certified.

E1: The system must maintain the list of relations specified in rule C2, and must ensure that the only manipulation of any CDI is by a TP, where the TP is operating on the CDI as specified in some relation.

E2: The system must maintain a list of relations of the form: (UserID, TPi, (CDIa, CDIb, CDIc, ...)), which relates a user, a TP, and the data objects that TP may reference on behalf of that user. It must ensure that only executions described in one of the relations are performed.

C3: The list of relations in E2 must be certified to meet the separation of duty requirement.

E3: The system must authenticate the identity of each user attempting to execute a TP.

C4: All TPs must be certified to write to an append-only CDI (the log) all information necessary to permit the nature of the operation to be reconstructed.

C5: Any TP that takes a UDI as an input value must be certified to perform only valid transformations, or else no transformations, for any possible value of the UDI. The transformation should take the input from a UDI to a CDI, or the UDI is rejected.

E4: Only the agent permitted to certify entities may change the list of such entities associated with other entities: specifically, the entities associated with a TP. An agent that can certify an entity may not have any execute rights with respect to that entity.

Together, these nine rules define a system that enforces a consistent integrity policy.

The ultimate goal of the Clark and Wilson model is to stimulate the development of better security systems and tools in the commercial sector. The authors suggest that the first step toward this goal would be to develop a new set of criteria that would more clearly address integrity enforcement. Clark [1987] offers a first cut at such a set of criteria, and further refines the initial effort [Clark 1989].

5.1.4.2 Analysis of Clark and Wilson

The second goal of computer integrity was defined as maintaining the internal and external consistency of data. The Clark and Wilson model addresses this goal through the use of IVPs and TPs. The purpose of an IVP is to confirm that all of the CDIs in the system conform to the integrity specification at the time the IVP is executed. An example of an IVP is the accounting principle of an independent audit, in which the books are balanced and reconciled to the external environment. An IVP verifies not only internal consistency but also external consistency by periodically cross-checking internal data with the external reality that it represents.

TPs maintain internal consistency by taking the system from one valid state to another valid state. TPs are structured so that a user cannot manipulate data arbitrarily, but only in constrained ways that preserve or ensure the internal consistency of the data. In the accounting example, a TP would correspond to a double entry transaction. Double entry bookkeeping ensures internal data consistency by requiring that any modification of the books be composed of two parts, which account for or balance each other. For example, if a check is to be written (which implies an entry in the cash account), there must be a matching entry on the accounts payable account.

Separation of duty is another major part of the Clark and Wilson model that addresses the third goal of computer integrity: preventing authorized users from making improper modifications. This goal is achieved indirectly by separating all operations into several subparts and requiring that each subpart be executed by a different person. For example, the process of purchasing and paying for some item might involve subparts: authorizing the purchase order, recording the arrival of the item, recording the arrival of the invoice, and authorizing payment. The last step should not be executed unless the previous three are properly done. If each step is performed by a different person, improper modifications should be detected and reported unless some of these people conspire. If one person can execute all of these steps, then a simple form of fraud is possible in which an order is placed and payment made to a fictitious company without any actual delivery of items. In this case, the books appear to balance; the error is in the correspondence between real and recorded inventory.

The separation of duty method is effective except in the case of collusion among employees. While this vulnerability might seem risky, the method has proved very effective in practical control of fraud. Separation of duty can be made very powerful by thoughtful application of the technique, such as random selection of the sets of people to perform some operation, so that any proposed collusion is only safe by chance.

One of the nine rules defining this policy, rule E2, points out the major difference between the Clark and Wilson model and the Bell and LaPadula model. Whereas the lattice model of Bell and LaPadula defines access restrictions for subjects to objects, the Clark and Wilson model (like the Lipner implementation) partitions objects into programs and data, and rule E2 requires subject/program/data access triples. Access triples are used to address the first integrity goal and also to implement the separation of duties concept.

An access triple (or triple) is a relation of the form: (UserID, TPi, (CDIa, CDIb, CDIc, ...)). The system maintains a list of triples to control which persons can execute which programs on specified CDIs. Chen [1989] suggests that it is possible to combine a ``user-to-program'' binding (which is already required in [DOD 1985]) with a ``program-to-data'' binding to enforce the triple. According to Chen [1989], several systems are capable of accommodating these bindings. In addition to the type manager/ pipeline approach taken by Boebert [1985] and the category approaches suggested by Shockley [1988] and Lee [1988], the Resource Access Control Facility (RACF) and the Access Control Facility 2 (ACF2) also appear capable of supporting user/program and program/data bindings.

Initial examination of Chen's implementation of access triples indicates that it would fail. Consider a system with two users (S1, S2), two transformation procedures (TP1, TP2), and three constrained data items (CDI1, CDI2, CDI3). The system security officer decides that the following triples should be enforced: (S1,TP1,CDI1), (S1,TP2,CDI2), (S2,TP1,CDI2), (S2,TP2,CDI3). The following bindings would be required in order to implement these triples using the approach suggested in [Chen 1989]: (S1,TP1), (S1,TP2), (S2,TP1), (S2,TP2), (TP1,CDI1), (TP1,CDI2), (TP2,CDI2), (TP2,CDI3). The (S1,TP1) binding required to satisfy the first triple combined with the (TP1,CDI2) binding required to satisfy the third triple results in user S1 having access to data item CDI2 through program TP1; this extra triple, (S1,TP1,CDI2), is a violation of system integrity.

There is evidence to suggest that implementing triples directly on an actual system may be prohibitively expensive in terms of performance [Karger 1988], or that requiring triples may reduce design and implementation flexibility at the operating system level [Chen 1989]. At the same time, triples do provide a means for limiting the risk of authorized users making improper modifications. In fact, the Clark and Wilson model effectively addresses all three goals of computer system integrity; the question is whether this model can be implemented realistically.

5.1.5 BREWER AND NASH MODEL

5.1.5.1 Discussion of Brewer and Nash

The Brewer and Nash model [Brewer 1989] presents a basic mathematical theory which is used to implement dynamically changing access permissions. The model is described in terms of a particular commercial security policy, known as the Chinese Wall. The model is developed by first defining what is meant by a Chinese Wall, and then by devising a set of rules such that no person (subject) can ever access data (objects) on the wrong side of that wall. The Chinese Wall security policy can be most easily visualized as the code of practice that must be followed by a market analyst who is providing corporate business services. Such an analyst cannot advise corporations where he has ``insider knowledge'' about a competitor, but the analyst is free to advise corporations which are not in competition with each other, and also to draw on general market information.

The following basic elements are used to mathematically define the Brewer and Nash model:

S: the set of subjects s; the users, or any program that might act on their behalf;

O: the set of objects o; files in which items of information are stored;

X(o), Y(o):functions which determine respectively the x and y components of the security label for a given object o;

xj, yj: X(oj) and Y(oj) respectively;

N: a boolean matrix with elements N(v,c) corresponding to the members of SxO; the elements N(v,c) take the value true if subject sv has, or has had, access to object oc or the value false if sv has not had access to object oc;

In the Brewer and Nash model, all corporate information is stored in a hierarchically arranged filing system where there are three levels of significance:

a At the lowest level, individual items of information, each concerning a single corporation, are called objects.

b) At the intermediate level, all objects which concern the same corporation are grouped together into a company dataset.

c) At the highest level, all company datasets whose corporations are in competition are grouped together into a conflict of interest class.

Associated with each object oj is the name of the company dataset yj to which it belongs and the name of the conflict of interest class xj to which that company dataset belongs.

The basis of the Chinese Wall policy is that people are only allowed access to information which is not held to conflict with any other information that they have accessed previously. In this approach, a subject is initially given complete freedom to access any object he cares to choose. For example, if there were three conflict of interest classes with the following member datasets (a, b, c) (d, e, f) and (g, h, i), a subject would initially be allowed access to all datasets a through i. If the subject then accessed an object in dataset e, the subject subsequently would be denied access to datasets d and f, while still having access to datasets a through c and g through i. If the subject next accessed an object in dataset a, the subject would be denied access to datasets b and c as well as d and f, leaving datasets a, e, g, h, and i accessible.

Two rules, the simple security rule and the *-property, are devised so that no subject can access objects on the wrong side of the Wall. The simple security rule formally defines that read access to any object or by any subject su is granted if and only if for all N(u,c) = true (i.e., su has had access to oc):

$$((yc = yr) \text{ or } (xc <> xr))$$

In other words, read access is only granted if or is in the same company dataset y as an object already accessed by that subject (i.e., within the Wall) or if or belongs to an entirely different conflict of interest class x. Once a subject has read an object, the only other objects readable by that subject lie within the same company dataset or within a different conflict of interest class. At most, a subject can have read access to one company dataset in each conflict of interest class.

The *-property is based on the concept of sanitized information defined by the authors. Sanitization takes the form of disguising a corporation's information, in particular, to prevent the discovery of that corporation's identity. If sanitization can be accomplished to prevent backward inference of origin, the sanitized information can be generally compared with similarly sanitized information relating to other corporations. Formally, for any object os, ys = yo implies that os contains sanitized information, and ys <> yo implies that os contains unsanitized information.

Thus, the *-property formally defines that write access to any object ob by any subject su is permitted if and only if N'(u,b) = true and there does not exist any object oa (N'(u,a) = true) which can be read by su for which:

ya <> yb and ya <> yo

In other words, write access is only permitted if access is permitted by the simple security rule, and if no object can be read which is in a different company dataset to the one for which write access is requested and contains unsanitized information. Thus, the flow of unsanitized information is confined to its own company dataset; sanitized information may however flow freely throughout the system.

5.1.5.2 Analysis of Brewer and Nash

This Brewer and Nash model was introduced in the context of commercial confidentiality requirements. We believe that a reinterpretation of this policy to address integrity issues would be both productive and straightforward. This model could be used in cases in which being able to access and make changes to two objects would allow a subject to manipulate the information in these two objects to fraudulent ends.

For example, suppose a consultant works for two competing companies, Ace and Acme, which both store information on their designs in a centralized DoD database. If the consultant has knowledge of both company's products, the consultant could subtly modify the design for Ace's product such that it had poor electromagnetic compatibility with Acme's product. Even though it worked at other times, Ace's product would fail in unexplained ways when used in the integrated weapons system in which Acme's product also was used. Ace's product would function correctly during testing, but would be perceived to be faulty in comparison to Acme's product when actually put into use.

There would be no evidence that this failure had been the result of an integrity violation in the design database; it would likely be perceived that Ace simply had done an inadequate job in the design of their product, or just that their product tended to fail a lot in the field. This type of conflict of interest problem was the origin of the Chinese Wall policy, although the original application was in the financial community. The Chinese Wall policy would prevent the consultant from accessing both companies' information. Thus, it addresses the third goal of integrity - preventing authorized users from making improper modifications.

The Chinese Wall policy was presented by Brewer et al. [1989] at the 1989 IEEE Symposium on Security and Privacy as an example of a policy which it was felt could not be addressed by the TCSEC requirements. As such, it is productive to identify whether proposed TCSEC revisions add the functionality needed to implement this policy. As was pointed out by John McLean in discussion following Brewer's presentation, the policy can in fact be implemented using categories if the user's allowed category set is permitted to change dynamically, and if this ability is used to vary the category set as a result of user accesses in accordance with the Chinese Wall policy.

Thus, it appears that categories or a comparable access partitioning mechanism are sufficient to implement the Chinese Wall, if a mechanism is also provided to change the user's allowed category set in response to user accesses. This feature is not necessarily provided by existing TCSEC implementations, since the TCSEC neither requires nor precludes support for dynamically changing category sets. Security policies which prohibit changes in mandatory access permissions for active processes as a result of a Tranquillity Principle that requires subjects to be inactive in order for their security labels to change, will not be able to support the Chinese Wall.

5.1.6 SUMMARY OF MODELS

Five models have been developed which suggest fundamentally different ways of achieving computer integrity. Biba [1977] was the first model to address integrity in a computer system, and it has the most established base of research because it is based on the Bell and LaPadula model for confidentiality. This model emphasizes the use of hierarchical levels. There are still no criteria for determining integrity levels and categories, so the Biba model does not seem to be the best approach for integrity. Goguen [1982] notes the advantages of domain separation and automaton-based state transformations. Capabilities would seem to be the most effective mechanism for implementing this model; the question is whether capability systems can be modified to control propagation, review, and revocation of access privileges. Sutherland [1986] addresses the effect of covert channels on integrity. The Sutherland model is primarily involved with representing the goal of protection rather than with a means of enforcing it. Clark and Wilson [1987] introduce the concept of access control triples that can be used to effectively implement separation of duties. This model was designed to meet commercial data integrity needs, but it can also be used in meeting integrity needs of secure military systems. Brewer [1989] presents a mathematical theory that implements dynamically changing access permissions.

Each of the models described in this chapter addresses one or more of the integrity goals defined at the beginning of the paper. All of the approaches address the first and most basic goal of computer system integrity. The second goal is addressed by the Clark and Wilson model, and the third goal is addressed by the Clark and Wilson and the Brewer and Nash models.

5.2 5.2 INTEGRITY MODEL IMPLEMENTATIONS

In this paper, implementations suggest realistic approaches to the theoretical basics identified in different models. While models describe what types of mechanisms are necessary to satisfy an integrity policy, implementations describe how specific mechanisms can be used together in a system to achieve integrity protection required by a particular security policy. Six methods have been identified as proposed implementations of one or more of the fundamental models described in the previous appendix: Lipner, Boebert and Kain, Lee and Shockley, Karger, Jueneman, and Gong. Lee and Shockley are two independently developed studies that are counted as one as they take essentially the same approach. The Boebert and Kain implementation is the only one we have been made aware of that is being actively pursued as a worked example.

5.2.1 LIPNER IMPLEMENTATION

5.2.1.1 Discussion of Lipner

Lipner [1982] examines two ways of implementing integrity in commercial data processing systems. The first method uses the Bell and LaPadula security lattice model by itself. The second method combines the Bell and LaPadula model with Biba's integrity lattice model.

Lipner's approach requires looking at security requirements in a different way from the prevalent view taken in the national security community. In particular, non-hierarchical categories are considered more useful than hierarchical levels. While a company will typically have organizational divisions (that correspond to categories), there will almost never be a notion of individuals having a clearance (that corresponds to a security level). Almost all employees work at the same level; they just do different things within that level. The key is to appropriately define a set of access classes, user types, and file types.

In Lipner's use of the Bell and LaPadula model, each subject and each object is assigned an access class. An access class is composed of one hierarchical classification level (e.g., Top Secret, Secret, Confidential, Unclassified) and one or more non-hierarchical, descriptive categories (e.g., NATO, Intelligence). In forming a lattice implementation to meet integrity requirements, the initial step is to define a set of access classes appropriate to achieve the desired restrictions.

In Lipner's first method, two levels distinguish between system managers and all other users, and four categories describe the types of work that are performed. Several user types and file types are assigned different levels and categories to implement integrity. Since almost all subjects and objects are assigned the ``System-Low'' level, categories are the most important part of the access class. By appropriately defining and assigning the categories, programmers, users, and system programmers are each limited to their own sphere of activity.

In the second method, Biba's integrity model is added to the basic (security) lattice implementation. The second method, like the first, assigns levels and categories to subjects and to objects. Its objective, however, is to prevent the contamination of high-integrity information by the infusion of lower-integrity data and by processing with lower-integrity programs. Integrity levels are provided to prevent the modification of system programs; integrity categories are used to separate different environments (e.g., development and production). This approach is used to simplify and render more intuitive a lattice policy application. It also limits the possibility of introducing a ``Trojan horse.''

5.2.1.2 Analysis of Lipner

Lipner's implementation of the Bell and LaPadula and Biba models is the first work to suggest that non-hierarchical categories are more important for integrity than hierarchical levels. Categories are used to address the first goal of integrity. Lipner took a survey of AIS auditors and AIS security officers to find that, in general, commercial users do not write their own programs and commercial program developers do not use the programs they write. Thus, users should not be ``above'' programmers, and programmers should not be ``above'' users; both types of users are at the same ``level,'' but should be completely separated from one another. Therefore, the appropriate use of categories is more important than the use of levels.

Lipner also appears to be the first person to separate objects into data and programs. This distinction is important because it is the first step toward addressing the third integrity goal (Clark and Wilson [1987] expand on this concept). Programs can be either passive (when they are being developed or edited) or active (when they are being invoked on behalf of a particular user), while data items are always passive. Programs are the means by which a user can manipulate data; thus, it is necessary to control which programs a user can execute and which data objects a program can manipulate.

Lipner's emphasis on categories and his separation of programs and data are positive steps forward. However, this approach has several drawbacks:

a Controls are at the level of user types and file types as opposed to being at the granularity of individual users and files; this is also a disadvantage of the Bell and LaPadula and Biba models. This drawback means that the granularity of control is inadequate to enforce access control triples.

b) Lipner recognizes that data should be manipulated only by certified (production) programs, but his controls do not provide a way to control what data a particular program can manipulate; there is no program-to-data binding explicitly called for.

c) Lipner's first method results in an ``all-powerful'' system manager who could conceivably manipulate the system at will. An all-powerful member of a computer system is always a risk because the least-privilege principle is not enforced in such a situation.

d) Lipner's second method is complex and will be difficult to administer.

The last item requires some explanation. Assume we have a system with two confidentiality levels (HiConf, LoConf) and two integrity levels (HiInteg, LoInteg); assume that proper category assignments exist for the sake of simplifying this explanation. The following combinations are possible:

a User1: HiConf, HiInteg

b) User2: HiConf, LoInteg

c) User3: LoConf, HiInteg

d) User4: LoConf, LoInteg

The Bell and LaPadula and Biba models use the operating system-oriented concepts of read and write. These can also be generalized to communications-oriented concepts of receive and send respectively. Bell and LaPadula requires that there be ``no read up'' and ``no write down'' in confidentiality level; Biba requires that there be ``no write up'' and ``no read down'' in integrity level. Thus, User1 may read (LoConf, HiInteg) and (HiConf, HiInteg) objects and write (HiConf, LoInteg) and (HiConf, HiInteg) objects. User2 may read any object and write only (HiConf, LoInteg) objects. User3 may read-only (LoConf, HiInteg) objects and may write any object. User4 may read (LoConf, LoInteg) and (LoConf, HiInteg) objects and write (LoConf, LoInteg) and (HiConf, LoInteg) objects.

Notice that User1 would be allowed to read any object if there were no integrity levels, but that the HiInteg level prevents him from reading half of the object types. Similarly, User 4 would be allowed to read any object if there were no confidentiality levels, but the LoConf level prevents him from reading half of the object types. The point is that confidentiality and integrity levels have an effect on one another; understanding this interaction and appropriately assigning levels to both users and objects is complex and will be difficult to administer.

5.2.2 BOEBERT AND KAIN IMPLEMENTATION

5.2.2.1 Discussion of Boebert and Kain

Boebert and Kain [Boebert 1985, Boebert 1988] introduce an implementation of the Goguen and Meseguer model using the concept of assured pipelines in the LOCK (LOgical Coprocessor Kernel) machine-formerly known as the Secure Ada Target (SAT). Assured pipelines provide a mechanism for ensuring that data of particular types can only be handled by specific trusted software. An assured pipeline is a subsystem that satisfies three properties: 1) it cannot be bypassed, 2) its actions cannot be undone, and 3) its actions must be correct.

The LOCK reference monitor system, SIDEARM, checks every individual access attempt for consistency with the security policy being enforced. The reference monitor is implemented in hardware, and resides between the processor, which generates memory access requests, and the memory system, which satisfies these requests. The LOCK reference monitor is actually a combination of a memory management unit (MMU), which has conventional rights checking facilities (read, write, etc.), and a tagged object processor (TOP), a new module responsible for the system's protection state. In particular, the TOP sets up the tables that define the access rights checked by the MMU. The link between the two of them is the MMU tables and the highly privileged code that sets the tables based on outputs from the SIDEARM [Boebert 1990].

Security attributes are associated with both subjects and objects, and the TOP must make appropriate comparisons between these security attributes to establish proper access rights in the MMU. Three security attributes are associated with subjects and three attributes are associated with objects. First, both subjects and objects have security (confidentiality) levels. Each subject performs its function for some ``user,'' whose identity is the second subject security attribute. The corresponding object's second security attribute is its access control list (ACL), which lists those users who are allowed access to the object's contents, along with the maximum access rights that each designated user is permitted. The third subject security attribute is the ``domain'' of its execution, which is an encoding of the subsystem of which the program is currently a part. The corresponding object's third security attribute is the ``type'' of the object, which is an encoding of the format of the information contained within the object.

The process of determining the access rights to be accorded a particular subject for access to a particular object uses all three security attributes. To enforce the mandatory access policy, the TOP first compares security levels of the subject and of the object, and computes an initial set of access rights according to the algorithm defined in Section 4.1.1.4 of the TCSEC.

To enforce the discretionary access policy, the TOP then checks the ACL for the object; the ACL entry that matches the user portion of the subject's context is compared against the initial set of access rights from the mandatory policy computation. Any access right in the initial set which does not appear in the ACL is deleted from the initial set. The result of this second determination check is an intermediate set of access rights.

The third LOCK access rights determination check compares the subject's domain against the object's type. Each domain is itself an object, and one of its attributes is a list of the object types accessible from the domain and the maximum access rights permitted from the domain to each type. Conceptually, the aggregation of these domain definition lists constitutes a table, which is called the Domain Definition Table (DDT). To make the domain-type check, the TOP consults the DDT row for the executing domain, finds the column for the object's type, and compares the resultant entry against the intermediate set of access rights. Any right in the intermediate set which does not appear in the DDT entry is dropped, and the result is the final set of access rights, which is transmitted to the MMU.

Domain changing may occur as a side effect of a procedure call. If the called procedure is not executable within the caller's domain, either the call is illegal or a domain change is necessary to complete the call. Information concerning domain changes is stored in a Domain Transition Table (DTT), which is stored as a set of lists associated with the calling domain. The LOCK system creates new subjects to handle domain changes, as required. When a call requires a domain change, LOCK suspends the calling subject and activates the called subject. The called subject has a different execution context, name space, and access rights, all of which will prevail for the duration of the procedure's execution.

5.2.2.2 Analysis of Boebert and Kain

Boebert and Kain's implementation is an object-oriented approach that addresses the first goal of integrity by focusing more on isolating the action than isolating the user. In other words, domains restrict actions to being performed in only one place in only one way; if you don't have access to that domain, you can't perform that action. This approach should help in preventing users from gaining unauthorized access through non-obvious paths.

This implementation is based on the Goguen and Meseguer model. The possible states are defined by the DDTs, and the state transitions are defined by the DTTs. Thus, integrity is achieved by carefully controlling these tables; this appears to be an achievable goal, although storage and management of the large number of tables may be complex. The Boebert and Kain approach is the only model ``implementation'' that is actually being implemented on a real system (LOCK). Their reported experience to date has shown that least privilege can be achieved without affecting system speed. The LOCK team is currently developing tools to aid in DDT construction and assurance [Boebert 1990].

Boebert and Kain's implementation provides a great deal of flexibility. In order to perform a certain action (e.g., modify an account balance), the user must have access to the appropriate domain. The domain may then further restrict what specific programs and data files the user can access (e.g., record a deposit for any account other than the user's, and only delete from the user's account). However, this flexibility may make the system more difficult to manage; some of the features may have to be restricted to ensure that unintentional errors are not made which jeopardize integrity (e.g., the user may be added to another account group that allows him to record deposits to his own account).

5.2.3 LEE AND SHOCKLEY IMPLEMENTATIONS

5.2.3.1 Discussion of Lee and Shockley

Independently, Lee [1988] and Shockley [1988] developed implementations of the Clark and Wilson model using Biba integrity categories and (partially) trusted subjects. Both implementations are based on a set of sensitivity labels. This set is built from two essentially independent components: every label represents a sensitivity with respect to disclosure, and an independent component representing a sensitivity with respect to modification. The labels corresponding to disclosure restrictions are not discussed; here, only the integrity labels are discussed in detail.

Using Lee's notation, a trusted subject, S, has two integrity labels, view-minimum and alter-maximum, denoted v-min(S) and a-max(S). Every object has one integrity label, label(O). An integrity label is simply a set of integrity categories; the notion of a hierarchical integrity level is not useful in this context. An integrity category can be interpreted as the name of a particular type of protected data. A subject, S, may write into an object, O, only if label(O) is a subset of a-max(S); S may read O only if either v-min(S) or a-max(S) is a subset of label(O). Any subject S with v-min(S) = a-max(S) is an untrusted subject.

Data is manipulated by ``certified transactions'' (TP in Clark and Wilson), which are partially trusted subjects. A partially trusted subject is allowed to transform data from a limited set of input types to a limited set of output types: it can read data that is marked with at least one of the categories specified in the subject's view-minimum label and still be allowed to write into data containers with any or all of the types in its alter-maximum label.

5.2.3.2 Analysis of Lee and Shockley

In addition to the two integrity labels, v-min(S) and a-max(S), every subject, S, also has two disclosure labels, v-max(S) and a-min(S). Each object has a disclosure label and an integrity label, which are made up of one or more disclosure and integrity categories respectively. Two requirements must be met for a subject to write to an object. First, the object's disclosure label must dominate a-min(S)-no write down with respect to disclosure. Second, a-max(S) must dominate the object's integrity label-no write up in integrity. The subject similarly may not read up in disclosure-v-max(S) must dominate the object's disclosure label-or read down in integrity-the object's integrity label must dominate v-min(S). These views address the first integrity goal.

The views provide subject-to-program, subject-to-object, and program-to-object bindings, which is sufficient to control which program a subject can invoke to manipulate a particular object. However, views in general bind only classes-or types-of subjects, programs, and objects, whereas the primary access control mechanism of the Clark and Wilson model (triples) binds individual users, programs, and objects.

(footnote #2) Thus, while Lee and Shockley demonstrate a transformation of the Clark and Wilson model to a lattice-based implementation, the actual operation of the such an implementation would likely prove to be cumbersome. Shockley also states that additions need to be designed and coded for building and checking tables of user/ program/data triples.

[1] It should be noted that the Shockley implementation does provide features for control based on individual, identified programs.

The major difficulty with the Lee and Shockley implementations seems to be management of large numbers of categories identified by Karger [1988]. Shockley notes that the Gemini Secure Operating System (GEMSOS) TCB has 90 bits available to represent access restrictions. The interpretation of these bits is confined to a single internal module that is easily modified. Although GEMSOS can encode a large number of categories (data types), managing these categories (storing all of the different combinations and performing the dominance checks) was not discussed and would seem to be a potential bottleneck.

The Lee and Shockley implementations do make three important contributions. First, categories can be interpreted as data types; thus, strong typing is important for implementing integrity. Second, a transaction-oriented approach seems best; thus, database technology should be very relevant. Finally, non-hierarchical categories alone are sufficient to implement the desired non-discretionary component of a Clark/ Wilson policy; thus, Biba's hierarchical levels are unnecessary. In this case, an important question to ask is, will Bell and LaPadula's hierarchical lattice fit with this non-hierarchical category approach? It is our conclusion that it will-the hierarchical disclosure part should be able to be added to the non-hierarchical categories as an additional restriction on access. In other words, assuming all of the sensitivity label requirements were satisfied, a hierarchical disclosure level would just be one more restriction on accessing an object.

5.2.4 KARGER IMPLEMENTATION

5.2.4.1 Discussion of Karger

Karger [1988] proposes an implementation of the Clark and Wilson model which combines SCAP, his secure capability architecture [Karger 1984], with the lattice security model. In this scheme, a capability-based protection kernel supports ACLs (representing the security lattice) at the outer level. In a typical capability system, the processor automatically loads capabilities into a cache for quick reference. Karger's proposal is to cause a fault or trap to the most privileged software domain (i.e., the security kernel) so that this privileged domain can evaluate whether the lattice model permits the capability to be used. Once a capability has been evaluated, it is placed in the cache so that it does not have to be reevaluated. If a lattice entry is changed, revocation can be achieved by flushing all of the capabilities for that object from the cache (causing new requests for the object to be freshly evaluated).

Karger suggests building CDIs out of abstract data types, with TPs as the operations of the type manager. Sealed (encrypted) SCAP capabilities can be used to implement abstract type management. As described above, SCAP can also intercept attempts to exercise capabilities by the TPs and require that special access control list checks be made to enforce separation of duty. Such an access control list for a TP would contain the Clark and Wilson triples.

As part of this implementation, audit trails form a much more active part of security enforcement than in previous systems. Karger introduces ``token capabilities'' to make the use of audit information easier. While taking the form of capabilities to prevent unauthorized tampering, token capabilities are in fact separate copies of individual audit records. They include both the name of the transaction and the name of the user who executed the transaction. The user's name must be recorded so that one can ensure that TP1 and TP2 are executed by different people. The name of the TP is recorded to prevent the token capability from being used more than once. Once used, the token capability is marked to prevent further use.

Token capabilities are used in conjunction with access control lists to ensure that permission to execute certain transactions or to modify certain CDIs can only be granted if certain previous transactions have been executed by specific individuals. Attempts to exercise regular capabilities cause an object's audit records (token capabilities) to be examined and compared with the appropriate ACL entry. If the request is determined to be legal, the regular capability is accepted and the desired action can be performed. Token capabilities simply provide a mechanism for making the proper audit records available for integrity policy decisions on a timely basis.

Entries in a separation of duty ACL are more complex than the simple ACLs supported in previous systems. Each entry consists of a boolean expression in which the first term is the name of the user who proposes to execute the action, and the other terms are token capabilities for required predecessor transactions. The boolean expression identifies the name and order of the preceding transactions that must be executed. As a final note, the author states that mechanism complexity and system performance will be problems regardless of the protection strategy that is adopted.

5.2.4.2 Analysis of Karger

By combining capabilities and access control lists, Karger greatly increases the implementation flexibility for achieving integrity. Karger requires that the ACLs contain the Clark and Wilson triples. This requirement by itself provides enough functionality for a system designer to implement static separation of duties (addressing both the first and third goals of integrity). Karger does not say how he would implement the triples or whether current systems would be sufficient; this is an important issue that must be addressed in future work.

The other part of Karger's implementation is the use of capabilities in support of the ACLs. Capabilities are used to provide domain separation so that actions are limited to particular domains, and also for quicker access. When a new capability is presented for invoking a certain program on a particular object, a trap to the security kernel is made to check the appropriate ACL entry. If the request is legal, the capability is stored in a cache to make subsequent accesses much faster.

Karger's complex ACLs not only contain triples, but they also specify the order in which transactions must be executed. These enhanced ACLs are used along with audit-based token capabilities to enforce dynamic separation of duties. Token capabilities can also be used to enforce well-formed transactions. The history of actions contained in these capabilities can be used to implement two-phase commit protocols. The two-phase commit protocol requires log records of all actions to ensure that any transaction either completely succeeds or does not occur at all. These log records are exactly the same records needed for making security policy decisions.

Notice that this implementation allows three levels of protection. First, triples implemented in the ACLs allow for basic integrity (static separation of duties). Second, capabilities can be used to support these ACLs to provide faster access and domain separation. Finally, enhanced ACLs and token capabilities support both dynamic separation of duties and well-formed transactions.

The additional flexibility provided by this implementation also creates some problems. Enhanced ACLs are much more complex than the ACLs implemented on current systems; thus, they may be more difficult to develop and to maintain. Token capabilities will add overhead that may become overly burdensome on system performance. Also, adding new applications may have a significant effect on existing separation of duty ACLs. Most importantly, future research needs to develop an efficient way of implementing access control triples.

5.2.5 JUENEMAN IMPLEMENTATION

5.2.5.1 Discussion of Jueneman

Jueneman [1989] proposes a defensive detection scheme that is based on mandatory and discretionary integrity controls, encryption, checksums, and digital signatures. This approach is intended for use on dynamic networks of interconnected Trusted Computing Bases (TCBs) that communicate over arbitrary non-secure media. Mandatory Integrity Controls prevent illegal modifications within the TCB, and detect modifications outside the TCB. Discretionary Integrity Controls are useful in supplementing the mandatory controls to prevent the modification, destruction, or renaming of an object by a user who has the necessary mandatory permissions, but is not authorized by the owner of the object. Encryption is used by the originator of an object to protect the secrecy or privacy of information. Checksums provide immutability and signatures provide attribution to allow the recipient of an object to determine its believability. The originator of an object should be responsible for assuring its confidentiality. The recipient should be responsible for determining its integrity.

It is assumed that the TCBs provide the equivalent of at least a B2 level of trust in accordance with the TCSEC. The B2 level provides mandatory and discretionary access controls and labels, process isolation achieved through the provision of distinct memory spaces under the control of the TCB, and the use of memory segmentation and read/write protection. In addition, in order to support the concept of separation of duty, the ability to exclude individuals or groups of individuals from accessing specified objects is assumed-a B3/A1 feature. Embedded cryptographic functionality is also assumed.

This implementation enforces mandatory and discretionary integrity controls across a computer network through the use of integrity labels. Two concepts are basic to an integrity label: integrity domain and integrity marking. Jueneman defines an integrity domain as the set of allowable inputs to a program (process) together with syntactic and semantic rules used to validate or reject those inputs and optionally produce one or more outputs. This use of the term ``domain'' is different from the definition used in both the TCSEC and the Boebert and Kain implementation, i.e., the set of objects that a subject has the ability to access.

An integrity marking has both hierarchical and non-hierarchical components. A hierarchical integrity level is an estimated probability that the process which created an object did so in accordance with the rules of a particular integrity domain or domains. Combining two or more integrity domains, the probabilities are assumed to be independent and are multiplied together to get an overall level (probability). A non-hierarchical integrity category qualifies the hierarchical integrity level; it is a shorthand representation of identifying the rules of the integrity domain that applied to that subject or object.

Integrity labels are incorporated as header and trailer labels within encrypted subject information or encrypted objects themselves. Subjects, data files, and programs each have different integrity labels. The particular elements included in these label types are discussed below.

5.2.5.1.1 Subject Integrity Label

A subject's integrity label contains a readclass, a writeclass, and a digital signature. A readclass is the range from the minimum integrity-read-limit to the maximum security-read-limit; a writeclass is the range from the minimum security-write-limit to the maximum integrity-write-limit. The read and write limits are made up of integrity markings.

A digital signature is a means by which a particular user or TCB digitally ``signs'' for the validity of objects it modifies. It provides non-repudiation protection of both the authorship of an object and the object's contents. Once a user has modified an object, he can digitally sign the object to unambiguously indicate his willing and conscious approval of the results. A receiver may require the originator's digital signature before an object will be processed.

5.2.5.1.2 Data File Integrity Label

The integrity label for a data file contains an integrity marking, a cryptographic checksum of the entire contents of the file, the name and cryptonym (digital signature) of the originator, and a bottom-up system of digitally signed certificates. A checksum is a separable code that is added to a block of data to help detect errors [Johnson 1989].

When original data is created, an extra piece of information, called the checksum, is appended to the block of data. The checksum is then regenerated when the data is received at its destination; the regenerated checksum and the original checksum are compared to determine if an error has occurred in the data, the checksum generation, the checksum regeneration, or the checksum comparison. Once a checksum has been computed, the original data and the check-sum are encrypted to produce a cryptographic checksum.

The digitally signed certificates are actually individual audit records. Each audit record has a ``pedigree'' part and a (optional) ``provenance'' part. The pedigree records which users ran what processes against what data inputs in order to produce given output(s); it is digitally signed by a TCB to show the TCB's approval. The provenance contains ancillary information (documentation, specifications, test cases and results, etc.); it is digitally signed by a user to show the user's intent.

5.2.5.1.3 Program Integrity Label

A program's integrity label contains all of the elements listed for a data file, and also includes the program's integrity domain.

5.2.5.2 Analysis of Jueneman

Jueneman makes a very good point when he says that ``it is almost impossible to discover or stop all [illegal actions] by [malicious] programs'' (e.g., virus, Trojan horse, unintentional errors in any systems or applications program). Therefore, his approach is to use a defensive containment mechanism that can at least detect any change to a subject or object, and that cannot itself be subverted, in order to prevent such programs from affecting the rest of the system. This approach itself seems to be a very appropriate type of design philosophy for integrity: detection instead of prevention.

Jueneman chooses to implement this design philosophy with some mechanisms that appear impractical, and some mechanisms that appear quite effective. His use of hierarchical probabilities and qualifying, non-hierarchical rules (categories) for achieving TCB integrity is complex and would be difficult to implement. It would be difficult to assign a consistent numerical value as to whether certain rules were followed (other than 0 or 1), and developing appropriate rules would be slow and application specific at best. Thus, it would seem more reasonable to use another implementation for TCB integrity.

However, Jueneman's use of encryption, checksums, and digital signatures seems to be an excellent proposal for addressing the first integrity goal at the distributed system level. Significant work has already been done in these three areas. Encryption, checksums, and digital signatures are well understood, and they add necessary integrity protection for distributed systems. To re-emphasize Jueneman's approach: the originator of an object is responsible for assuring its confidentiality, and the recipient is responsible for determining its integrity.

5.2.6 GONG IMPLEMENTATION

5.2.6.1 Discussion of Gong

Gong [1989] presents the design of an Identity-based CAPability protection system (ICAP), which is aimed at a distributed system in a network environment. ICAP merges the ACL approach and the capability approach; ACLs support a capability protection mechanism (just the opposite of Karger's SCAP), and nicely solve the problem of revocation. Gong's design provides support for administrative activities such as traceability. This approach also works for a centralized system. Compared with existing capability system designs, ICAP requires much less storage and has the potential of lower cost and better real-time performance.

A classic capability is represented as

$$(Object, Rights, Random)$$

in which the first item is the name of the object and the second is the set of access rights. The third is a random number to prevent forgery and is usually the result of a one-way function f, *Random = f(Object,Rights)*

Here, f can be a publicly known algorithm. It should not be based on other secret keys because key distribution introduces other difficulties. Its requirements are that it is computationally infeasible to inverse f and, given a pair of input and matching output, it is infeasible to find a second input which gets the same output. When an access request arrives at the object server together with a capability, the one-way function f is run to check the result against the random number to detect tampering. If the capability is valid, the access is granted to the subject.

In a classic system, a capability is created for each different set of access rights required for each object, and the capabilities that are kept by the server and other subjects for the same set of rights are the same. For example, if subject S1 possesses a read-only right and S2 possesses a read and write right for an object, the server has to have two different capabilities, C1 and C2. S1 holds C1 and S2 holds C2.

In ICAP, only one capability for each object is stored at the server, and different subjects' capabilities for the same object are distinct. This improvement in storage is achieved by changing the semantics of those items in traditional capabilities to incorporate identities, e.g., the owner-id, into the capabilities. When the server creates a new object on behalf of a subject S1, an ``internal'' capability is created as

(Object,Random0)

and stored in the server's internal table. As usual, this table is protected against tampering and leakage. S1 is sent an ``external'' capability

(Object,Rights,Random1)

which looks exactly the same as a classic capability but

Random1 = f(S1,Object,Rights,Random0)

When S1 presents the capability later, the server runs the one-way function f to check its validity. Note the number Random0 should possess a kind of freshness to counter a playback attack. For example, it could have a timestamp.

The server holds only one internal capability for each object, regardless of how many different subjects have access to a particular object. Thus, there is a reduction in the amount of storage required at the server. Subjects continue to hold distinct external capabilities for each object they are allowed to access.

When S1 wants to pass its external capability

(Object,Rights,Random1)

to a process owned by S2, it must explicitly present the request to the server. If the request complies with the security policy, the server retrieves the secret Random0 from its internal table, creates

(Object,Rights,Random2)

where

Random2 = f(S2,Object,Rights,Random0) and passes it to S2.

Because the internal random number is kept secret, external capabilities are protected against forgery. Moreover, since proper authentication is done, subjects cannot masquerade as others. Subject S2 cannot use S1's capability even if S2 possesses a copy of S1's capability because the identities are different, hence the results of applying f will be different; S2 must be given its own capability to access a particular object. Any valid propagation has to be completed by the server rather than by the subjects. In other words, the server can monitor, mediate, and record any capability propagation.

When a capability is to be propagated, the kernel or an ``access control server'', which may or may not be the object server, checks to see whether to allow the propagation, according to the security policy. The intuitive motive of this scheme is the observation that the number of capability propagations is usually much less than the number of their uses, so it seems more economic to check the security policy at propagation time than at access time. The object server does not check against the security policy when a capability is later used for access. Instead, exception lists and propagation trees are used to revoke access.

Exception lists are associated with each internal capability to achieve rapid revocation. When S1 wants to revoke a capability it gave S2 earlier, it presents the request to the server. The server then updates the corresponding exception list. When an access is required, both the exception list and the capability's validity are checked. These checks can be done in parallel.

Capability propagation trees are stored at the access control server, and they can be resolved in background mode to achieve complete revocation. For a capability passed from S1 to S2 and then to S3, the propagation tree entry would look like

(Object,Rights,S1,S2,S3,Random3)

where

Random3 = f(S1,S2,S3,Object,Rights,Random0)

When something goes wrong, it is straightforward to know from the access control lists who has what access to an object. In addition, it is easy to know from the propagation trees how the access rights were propagated.

5.2.6.2 Analysis of Gong

Gong's implementation is an effective, conceptually straight-forward implementation of access control lists supporting a capability-based protection system. His approach takes advantage of the domain separation and performance characteristics of capabilities, while maintaining control over propagation and revocation of access rights. Just as importantly, the structure of the implementation is simple and allows the incorporation of any disclosure and/or integrity policy. This implementation can be used to address the first and third goals of integrity.

The implementation is made up of users, objects, object server(s), and a centralized access control server. The access control server maintains ACLs that describe the desired protection policy, and it also monitors any new requests for capabilities. An object server maintains one internal capability for each object, and communicates with the access control server on behalf of the users. These two types of servers each contain additional data structures that Gong uses to solve the two main problems encountered in the past with capability-based protection systems: propagation and revocation. Propagation trees are stored on the access control server to keep track of which users have access to an object and how they obtained that access. Each object server stores an exception list that records when a user wants to revoke access to another user.

This implementation provides flexibility in several areas. The most significant accomplishment is that the type of protection policy is immaterial to Gong's implementation; it could be a Bell and LaPadula disclosure lattice, a Biba integrity lattice, Clark and Wilson triples, or Lee/Shockley non-hierarchical categories. Whatever the policy, it is incorporated as part of the access control server.

There are other aspects to Gong's implementation that allow for flexibility of use. First, domain IDs can be incorporated into user IDs, if necessary. Second, discretionary control can be added on top of the existing non-discretionary control structure. Third, sealed (encrypted) capabilities can be used to implement protected subsystems. Finally, the given capability structure can be expanded.

Gong's implementation has two additional advantages besides flexibility. First, the protection policy is checked at propagation time, which is more economic than checking the policy on each access. Second, although exception lists are very useful for achieving rapid revocation, complete revocation can be performed in the background by the access control server. This structure also allows revocation to be withdrawn. Since revocation is initially marked in the exception list, a false alarm or error can be resolved without invoking the expensive full revocation mechanism.

A disadvantage in the Gong implementation is that it can only implement static separation of duties because the ACLs are not checked at access time. This limitation may be a major drawback in the long run if dynamic separation of duties becomes an accepted and required part of system specifications.

5.2.7 SUMMARY OF MODEL IMPLEMENTATIONS

Based on one or more of the models we have discussed, seven implementations have been identified and analyzed. Lipner [1982] is the first work to emphasize the importance of non-hierarchical categories for achieving integrity. This implementation also introduces the important distinction between program objects and data objects. Boebert [Boebert 1985, Boebert 1988] presents a flexible, object-oriented approach that focuses more on isolating the action than isolating the user. Lee [1988] and Shockley [1988] introduce minimum and maximum views for controlling access to objects. This approach may be difficult to manage, but it does point out the usefulness of strong typing and transaction-based operations. Karger [1988] and Gong [1989] both combine the advantages of capabilities (speed and domain separation) and ACLs (review and revocation). Karger's approach uses capabilities in support of ACLs; Gong uses ACLs in support of capabilities. Karger provides more flexibility, but Gong's approach seems to be more realistic for use in an actual system. Jueneman [1989] has a highly complex approach for TCB integrity, but his use of encryption, checksums, and digital signatures seems to be an excellent proposal for promoting and preserving certain aspects of integrity in distributed systems.

Each of the implementations described in this section address one or more of the integrity goals defined at the beginning of the paper. All of the approaches address the first, and most basic, goal of computer system integrity. The second goal is not addressed directly by any of the implementations. The third goal is addressed by several implementations [Lee 1988, Shockley 1988, Karger 1988, Gong 1989] that recommend using concepts described in the Clark and Wilson model.

5.3 GENERAL ANALYSIS OF MODELS AND MODEL IMPLEMENTATIONS

Models and model implementations that protect information in computer systems are all based on some policy variation of the principle of separation. A common theme is to restrict access to information by separating more ``sensitive'' information from less important data, and concentrating the protection effort on the sensitive information. We have described several models and implementations which have different philosophies on the concept of separation. Now we look at the pros and cons of five separation philosophies: hierarchical levels, non-hierarchical categories, access control triples, protected subsystems, digital signatures, and encryption. We also discuss the combination of capabilities and ACLs, and conclude with a general analysis.

5.3.1 Hierarchical Levels

One of the earliest computer protection models was developed in 1975 by Bell and LaPadula, with the emphasis being to prevent unauthorized disclosure of data or information. Their approach to modelling separation was through a lattice, designed to neatly coincide with the existing governmental classification structure (e.g, Top Secret, Secret, Confidential, Unclassified). The Bell and LaPadula model was the inspiration for Biba's integrity model. Both models emphasize levels in establishing rules for information protection.

The current governmental classification system is oriented toward preventing improper disclosure of information; Bell and LaPadula developed their model with this classification system in mind. If a similar classification system were developed for preventing improper modification, the Biba model would be a good choice for representing these needs. Until then, hierarchical levels will not be very useful for computer system integrity.

5.3.2 Non-hierarchical categories

The term ``category'' is used to describe non-hierarchical separation. In 1982, Lipner expressed the need to have a different outlook on computer security. He recommended that primary emphasis be placed on categories, and only minor emphasis be placed on levels. Outside of the government, few organizations operate according to the concept of levels. Most companies have employees at the same ``level'', but with different responsibilities. Even if levels are implemented, categories must provide appropriate separation between users within each level. The concept of ``roles'' may be more accurate or applicable.

5.3.3 Access Control Triples

Lipner was the first person to identify the need for distinguishing between program objects and data objects. Clark and Wilson built on this distinction to establish the concept of access control triples. Clark and Wilson view triples as being necessary for implementing integrity because triples allow the explicit identification of which data a user is allowed to manipulate with a particular program. Triples also provide the ability to implement separation of duty, another necessary attribute of a computer system that has integrity.

Triples provide a useful paradigm for implementing integrity. While the Clark and Wilson model does not have all the details worked out, we believe that it is a useful guide for future research in addressing computer system integrity.

5.3.4 Protected Subsystems

Goguen and Meseguer introduced an automaton-based model that focuses on the transition from one valid state to the next as defined by the particular security policy being enforced. The emphasis of this model is to separate users' working spaces to provide non-interference and to prevent unauthorized access. Boebert and Kain's implementation is based on this concept of protected subsystems.

Protected subsystems implement a finer degree of separation within categories. They exclude certain access rights that a user has outside the subsystem. A user can only access certain objects from within the protected subsystem, and then only in constrained ways defined by the subsystem.

Strong typing is a promising area for implementing protected subsystems. Strong typing provides separation by limiting the number of entry points for executing certain programs. In other words, a user must have access to the appropriate type in order to execute the programs within that domain. However, strong typing does not limit the execution of specific programs once a user has entered the domain; that is the responsibility of access controls (i.e., triples).

5.3.5 Digital Signatures/Encryption

The last philosophy of separation is based on digital signatures and encryption. Jueneman is the biggest supporter of this approach; he strongly argues that digital signatures and encryption are the only way to provide protection across a distributed network. Jueneman may be correct in this matter, although the use of digital signatures and encryption appears to be a necessary rather than a sufficient mechanism for protection in distributed environments.

5.3.6 Combination of Capabilities and ACLs

Both Karger and Gong combine the advantages of capabilities (protection domains and performance) and ACLs (control over propagation and revocation of access rights) to provide protection. Karger's model allows for dynamic separation of duties, but Gong's model is simple, implementable, and more efficient for static separation of duties.

5.3.7 Summary of General Analysis

Having analyzed each of the individual models, some overall conclusions can be identified. First, as Lipner suggested, categories seem to be much more important than levels for implementing integrity. Also, it is important to distinguish between program and data objects because programs are the means by which a user modifies data objects. Second, the Clark and Wilson model appears to be a useful starting point for addressing the issues of integrity. In particular, access triples are recommended (if not required) to explicitly identify which programs a user may invoke to modify particular data objects; triples cannot be implemented with existing systems. Third, encryption and digital signatures look to be an effective way of maintaining integrity over a distributed network. Fourth, the combination of capabilities and access control lists provides maximum separation, efficiency, and access review. Finally, many of the models talk about validation and verification of the programs that carry out the actual manipulations (TPs in Clark and Wilson). Continued research needs to focus on methods for assuring that programs do what they are intended to do and nothing more.

6 CONCLUSIONS

This paper has discussed the need for integrity to be promoted and preserved with respect to data and systems. It has recognized that this need exists for military, public, private, and commercial organizations who depend on the integrity of their systems and their data in automated information processing, process control, and embedded computing applications. Further, it has shown that this need has been recognized since the early days of computer systems development, and that many of the protection mechanisms now routinely incorporated into today's computers were the result of addressing concerns of integrity. This latter point is important in that often the argument is made that we have had no worked examples of integrity and that we need to conduct a significant amount of research before any criteria are written. This paper tries to add some balance to that argument. The paper illustrates that there is a significant body of knowledge available about integrity and integrity mechanisms, and that such knowledge can be presented in a way to aid in initial formulations of criteria. The paper also recognizes that there are many aspects of integrity that require further investigation. It is the idea of concurrently pursuing both criteria and criteria-enabling research that we believe is key to making the rapid advances necessary in meeting the recognized needs for integrity.

6.1 SUMMARY OF PAPER

The paper discussed the difficulty of trying to provide a single definition for the term integrity as it applies to data and systems. We conclude that a single definition is probably not possible and, indeed, not needed. An operational definition that encompasses various views of the issue seems more appropriate. We offer such an alternative so that progress beyond definitional aspects can be made. Our framework, or operational definition, provides a means to address both data and systems integrity and to gain an understanding of important principles that underlie integrity. It provides a context for examining integrity preserving mechanisms and for understanding the integrity elements that need to be included in system security policies.

The paper has provided a set of principles that are important to integrity. We do not claim that the set is complete or applies to all systems or applications. We do believe that it is useful as basis for developing integrity policies and mechanisms. We encourage others to debate and to strengthen this set of principles.

We have shown that the promotion and preservation of data and systems integrity can involve a wide diversity of mechanisms. The mechanisms have been categorized to show that they serve a relatively small set of distinct purposes. Separation stands out as the most pervasive feature. We use the term ``policy'' to describe the administrative courses of actions which characterize a group of mechanisms in promoting or preserving integrity. We acknowledge that not all of these mechanisms are automated, but we believe that by including them we have given greater insight into what types of controls need to be provided and the types of threats which must be countered by automated integrity mechanisms. A significant number of these mechanisms serve to support human-enforced integrity; thus, without humans involve in the process, these mechanisms will not necessarily provide additional benefits. The costs and benefits of research required to automate some of these mechanisms should be evaluated. All of these mechanisms need to be examined in light of other protection goals, i.e., confidentiality, to ensure that any conflicting goals are identified and resolved prior to the inclusion of these mechanisms into systems that need to satisfy multiple protection goals.

The paper has provided an overview of several models and model implementations (paper studies) of integrity. These models are still rather primitive with respect to the range of coverage suggested by examining both data and systems integrity. No model, on its own, supports all of the integrity policies that we have identified. Of these models, the Clark and Wilson model now seems to be receiving the most research attention. It is not a formal mathematical model, and its lack of preciseness in its details still leaves many elements ambiguous. Although these ambiguities need to be resolved and many aspects empirically proven, the Clark and Wilson model provides a fresh and useful point of departure for examining issues of integrity. The Biba model addresses a more fundamental concept, that of contamination. While this model may seem initially appealing, it seems contrary to what is actually occurring in data processing where most of the mechanisms we apply are meant to increase the integrity of the data. Further, it does not seem to be practical or easily implementable, i.e., the determination of labels for specific data and the subsequent labelling operations of data items may be extremely difficult and of a much higher cost than the value returned. The Goguen and Meseguer model is related to integrity by the concept of non-interference, which can be used to model behaviors of active entities and to show that actions are separate and non-interfering.

6.2 SIGNIFICANCE OF PAPER

By going beyond attempting to define integrity, the document provides material that we believe will be useful to system designers, criteria developers, and to individuals trying to gain a better understanding of the issues of data and systems integrity. The study provides a framework and foundational material to continue the efforts toward developing criteria for building products which preserve and promote data and systems integrity. However, this study is only a beginning and remains incomplete in terms of fully addressing the topic. Further principles should be sought out. Additional mechanisms in the areas of fault tolerance, database management, communications, and in specific application areas should be described and analyzed.

The paper reinforces the realization that data and systems integrity are important and necessary, and that they need to be addressed in a determined and methodical manner. We conclude that there is a great deal known about certain aspects of data and systems integrity and, in many cases, that there exists a history of experiences with both the principles and the mechanisms. We also conclude that there are many things yet to be learned with respect to assembling this experience and knowledge into coherent systems, e.g., moving from manual mechanisms to automated mechanisms. One of the main conclusions that can be drawn from this paper is that mechanisms can be applied at various layers of system abstraction. This concept of layering needs to be examined to determine what, if any, interfaces need to exist between mechanisms at different system layers and what protocols need to exist within layers. IDA is undertaking a follow-on study to address the allocation and layering of mechanisms.

Integrity criteria need to be written. For some aspects, we conclude that there is sufficient understanding to write specific criteria, but for other aspects of such criteria, more experience, research, debate, and proofs of concepts will be needed. We believe that this partial knowledge should not delay the writing of criteria. Rather, we recognize the need to establish a means to make the criteria, and thus the systems, evolvable with respect to integrity protection. Establishing this means may require more participation by systems vendors in the evolutionary development of integrity criteria than there was in the development of confidentiality criteria. The key here is to understand what is involved in designing systems for evolution so that criteria do not unnecessarily stifle new system designs or new concepts for preserving or promoting integrity.

6.3 FUTURE RESEARCH

The following studies are suggestive of those that should be undertaken to extend and apply the principles, mechanisms, models, and model implementations that have been presented in this paper.

Allocation Study

The understanding of layering and allocating the various integrity mechanisms needs to be broadened. There are many questions that need to be addressed as part of this understanding. For example, how should the various mechanisms be allocated across the applications, operating systems, database management systems, networking systems, and hardware bases? What is the basis for this allocation? What are the interactions between and among layers? What are the concerns between allocation and system evolution? What might one expect from a vendor's product as a result of allocation? This study is critical to refining specifications for product-oriented control objectives that will form the basis of product criteria.

Interface and Protocol Studies

Where there appears to be a need for interaction between mechanisms in different system layers, the specific interfaces and communications protocol must be established. These studies should examine current interfaces and develop new ones as appropriate. They should design new or show how existing protocols can be used. These studies should examine the efficiency and effectiveness of potential interfaces and protocols. Where appropriate, these studies should recommend proofs of concept research.

Demonstration/Validation Studies

We need to have some worked examples to ensure that the arrangement of mechanisms is workable, that interface and protocol concepts are valid, and that criteria are testable. These studies should incorporate a variety of systems architectures to preclude development of criteria with a single or limited focus.

Criteria Development Study

We need to begin to develop criteria in parallel with the protocol and demonstration/validation studies. This effort should interact with these two areas in receiving and providing direction. One major part of the criteria study should be form, a second part should be scope and specific content, a third part should address the evolution of criteria, and a final part should address the linkages of product criteria to certification and accreditation of systems by using authorities.

REFERENCE LIST

[1] Abrams 1990

Abrams, Marshall D., Leonard J. La Padula, Kenneth W. Eggers, and Ingred M. Olson. 1990. A Generalized Framework for Access Control: An Informal Description. In Proceedings of the 13th National Computer Security Conference, 1-4 October 1990, Washington, D.C., 135-143. Gaithersburg, MD: National Institute of Standards and Technology.

[2] Biba 1977

Biba, K.J. 1977. Integrity Considerations for Secure Computer Systems. Bedford, MA: MITRE Corporation.

[3] Bishop 1979

Bishop, M. and L. Snyder. 1979. The Transfer of Information and Authority in a Protection System. In Proceedings of the 7th Symposium on Operating Systems Principles, 10-12 December 1979, Pacific Grove, California, 45-54. New York: Association for Computing Machinery.

[4] Boebert 1985

Boebert, W.E. and R.Y. Kain. 1985. A Practical Alternative to Hierarchical Integrity Policies. Proceedings of the 8th National Computer Security Conference, 30 September-3 October 1985, Gaithersburg, Maryland, 18-27. Gaithersburg, MD: National Bureau of Standards [now the National Institute of Standards and Technology].

[5] Boebert 1988

Boebert, W.E. 1988. Constructing an Infosec System Using LOCK Technology. In Proceedings of the 11th National Computer Security Conference, 17-20 October 1988, Baltimore, Maryland: Postscript, 89-95. Gaithersburg, MD: National Bureau of Standards [now the National Institute of Standards and Technology].

[6] Boebert 1990

Boebert, W.E. 1990. Electronic communication to authors.

[7] Bonyun 1989

Bonyun, David A. 1989. On the Adequacy of the Clark-Wilson Definition of Integrity. In Report of the Invitational Workshop on Data Integrity, January 25-27, 1989, Gaithersburg, Maryland, B.5B.5-9. Gaithersburg, MD: National Institute of Standards and Technology.

[8] Branstad 1989

Branstad, Martha, Homayoon Tajalli, Frank Mayer, and David Dalva. 1989. Access Mediation in a Message Passing Kernel. In Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy, May 1-3, 1989, Oakland, California, 66-72. Washington, D.C.: IEEE Computer Society Press.

[9] Brewer 1989

Brewer, David F. C. and Michael J. Nash. 1989. The Chinese Wall Security Policy. In Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy, May 1-3, 1989, Oakland, California, 206-214. Washington, D.C.: IEEE Computer Society Press. Oakland, CA

[10] Chen 1989

Chen, Tom. 1989. Operating Systems and Systems-Group 1 Report. In Report of the Invitational Workshop on Data Integrity, January 25-27, 1989, Gaithersburg, Maryland, 4.1-14.1-11. Gaithersburg, MD: National Institute of Standards and Technology. NIST Special Publication 500-168.

[11] Clark 1987

Clark, D.D. and D.R. Wilson. 1987. A Comparison of Commercial and Military Computer Security Policies. Proceedings of the IEEE Symposium on Security and Privacy, 184-194, Oakland, CA.

[12] Clark 1989

Clark, David D. and David R. Wilson. 1989. Evolution of a Model for Computer Integrity. In Report of the Invitational Workshop on Data Integrity, January 25-27, 1989, Gaithersburg, Maryland, A.2-1A.2-13. Gaithersburg, MD: National Institute of Standards and Technology. NIST Special Publication 500-168.

[13] Courtney 1989

Courtney, Robert H. 1989. Some Informal Comments About Integrity and the Integrity Workshop. In Report of the Invitational Workshop on Data Integrity, January 25-27, 1989, Gaithersburg, Maryland, A.1-1A.1-5.1. Gaithersburg, MD: National Institute of Standards and Technology. NIST Special Publication 500-168.

[14] DOD 1985

Department of Defense. 1985. DoD Trusted Computer System Evaluation Criteria. DoD 5200.28-STD. Washington, D.C.: U. S. Government Printing Office.

[15] Eichin 1989

Eichin, Mark W. and Jon A. Rochlis. 1989. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. In Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy, May 1-3, 1989, Oakland, California, 326-343. Washington, D.C.: IEEE Computer Society Press.

[16] Gligor 1987

Gligor, V.D., J.C. Huskamp, S.R. Welke, C.J. Linn, and W.T. Mayfield. 1987. Traditional Capability-Based Systems: An Analysis of Their Ability to Meet the Trusted Computer Security Evaluation Criteria. IDA Paper P-1935, available as NTIS AD-B119 332. Alexandria, VA.

[17] Goguen 1982

Goguen, J.A. and J. Meseguer. 1982. Security Policies and Security Models. Proceedings of the 1982 Berkeley Conference on Computer Security, 11-20. Berkeley, CA.

[18] Gong 1989

Gong, Li. 1989. A Secure Identity-Based Capability System. In Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy, May 1-3, 1989, Oak- land, California, 56-63. Washington, D.C.: IEEE Computer Society Press.

[19] ISO 1990

International Standards Organization/International Electrotechnical Commission Joint Technical Committee 1/Subcommittee 21 (ISO/IEC JTC1/SC21). 1990. Information Technology-Security Frameworks for Open Systems-Part 2: Authentication Framework. N 5281 (proposed Second CD 10181-2).

[20] Johnson 1989

Johnson, Barry W. 1989. Design and Analysis of Fault-Tolerant Digital Systems. Reading, MA: Addison-Wesley.

[21] Jueneman 1989

Jueneman, Robert R. 1989. Integrity Controls for Military and Commercial Applications, II. Falls Church, VA: Computer Sciences Corporation. CSC/PR-89/3001.

[22] Karger 1984

Karger, Paul A. and Andrew J. Herbert. 1984. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In Proceedings of the 1984 IEEE Symposium on Security and Privacy, April 29-May 2, 1984, Oakland, California, 2-12. Silver Spring, MD: IEEE Computer Society Press.

[23] Karger 1988

Karger, Paul A. 1988. Implementing Commercial Data Integrity with Secure Capabilities. In Proceedings of the IEEE Symposium on Security and Privacy, April 18-21, 1988, Oakland, California, 130-139. Washington, D.C.: IEEE Computer Society Press.

[24] Knight 1986

Knight, J.C., and N.G. Leveson. 1986. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. IEEE Transactions on Software Engineering 12 (January): 96-109.

[25] Lee 1988

Lee, Theodore M.P. 1988. Using Mandatory Integrity to Enforce ``Commercial'' Security. In Proceedings of the IEEE Symposium on Security and Privacy, April 18-21, 1988, Oakland, California, 140-146. Washington, D.C.: IEEE Computer Society Press.

[26] Lipner 1982

Lipner, S.B. 1982. Non-Discretionary Controls for Commercial Applications. Proceedings of the IEEE Symposium on Security and Privacy, 2-10. Oakland, CA

[27] Longley 1987

Longley, Dennis and Michael Shain. 1987. Data & Computer Security: Dictionary of Standards, Concepts and Terms. New York: Stockton Press.

[28] Mayfield 1991

Mayfield, Terry, John M. Boone, Stephen R. Welke. 1991. Integrity-Oriented Control Objectives: Proposed Revisions to the Trusted Computer Systems Evaluation Criteria (TCSEC), DoD 5200.28-STD. Alexandria, VA: Institute for Defense Analyses. IDA Document D-967.

[29] NCSC 1988

National Computer Security Center (NCSC). 1988. Glossary of Computer Security Terms. Washington, D.C.: U.S. Government Printing Office.

[30] Outposts 1989

Air Safety: Is America Ready to `Fly by Wire'? The Washington Post April 2, 1989. Outposts section: C3.

[31] Rivest 1978

Rivest, R.L., A. Shamir, and L. Adleman. 1978. A Method of Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM 21, 120-126 (February).

[32] Roskos 1984

Roskos, J.E. 1984. Data Movement, Naming, and Ambiguity. Nashville, TN: Vanderbilt University, Department of Computer Science. Technical Report CS-84-05.

[33] Saltzer 1975

Saltzer, J.H. and M.D. Schroder. 1975. The Protection of Information in Computer Systems. Proceedings of the IEEE 63, (September): 1278-1308.

[34] Saltzer 1977

Saltzer, J.H. 1977. Naming and Binding of Objects. In Operating Systems: An Advanced Course, 99-208. New York: Springer-Verlag.

[35] Schaefer 1989

Schaefer, M., W. C. Barker, and C. P. Pfleeger. 1989. Tea and I: An Allergy. In Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy, May 1-3, 1989, Oakland, California, 178-182. Washington, D.C.: IEEE Computer Society Press.

[36] Seecof 1989

Seecof, M. and R. Hoffman. 1989. A320/MD-11 F-B-W differ on pilot authority. Electronic posting to the Forum on Risks to the Public in Computers and Related Systems. Vol. 9, Issue 4 (July 13).

[37] Shockley 1988

Schockley, W.R. 1988. Implementing the Clark/Wilson Integrity Policy Using Current Technology. In Proceedings of the 11th National Computer Security Conference, 17-October 1988, Baltimore, Maryland, 29-37. Gaithersburg, MD: National Bureau of Standards [now the National Institute of Standards and Technology].

[38] Steiner 1988

Steiner, Jennifer, Clifford Neuman, Jeffrey I. Schiller. 1988. Kerberos: An Authentication Service for Open Network Systems. Cambridge, MA: Massachusetts Institute of Technology.

[39] Struble 1975

Struble, George E. 1975. Assembler Language Programming: The IBM System/360 and 370. Reading, MA: Addison-Wesley.

[40] Sutherland 1986

Sutherland, David. 1986. A Model of Information. In Proceedings of the 9th National Computer Security Conference, 15-18 September 1986, Gaithersburg, Maryland, 175-183. Gaithersburg, MD: National Bureau of Standards [now the National Institute of Standards and Technology].

[41] Webster 1988

Webster's Ninth New Collegiate Dictionary. 1988. Springfield, MA: Merriam-Webster, Inc.

APPENDIX : GENERAL INTEGRITY PRINCIPLES

Several general security concepts should be borne in mind while considering the mechanisms described in Section 4. These concepts are based primarily on a paper by Jerome Saltzer and Michael Schroeder which has formed the basis for many of the ideas in later computer security work, including the TCSEC. This appendix is intended to help give a perspective to the very diverse set of mechanisms reviewed in Section 4.

In their 1975 paper, ``The Protection of Information in Computer Systems,'' Saltzer and Schroeder address a number of issues related to computer security [Saltzer 1975]. While the focus of the paper was not exclusively integrity, many of the ideas and concepts presented in that paper relate to current integrity issues. In particular, the authors discuss ``Functional Levels of Information Protection'' and ``Design Principles,'' which will be reviewed in this appendix. In addition to being applicable to integrity, these principles relate directly to many concepts in the TCSEC. This paper compiles and draws on a large amount of early material dealing with computer security. References to those sources will not be noted here, but can be found in the original report.

1 TRADITIONAL DESIGN PRINCIPLES

The design principles presented by Saltzer and Schroeder are intended to reduce design and implementation flaws which lead to unauthorized disclosure, modification, or denial of system resources. The authors emphasize that these principles should not be taken as absolute rules, but that violations of the principles should be taken as potential sources of trouble and therefore should not be undertaken lightly. These principles tend to reduce both the number and severity of flaws.

1.1 ECONOMY OF MECHANISM

This principle states that the design, for all aspects of the system and especially for protection mechanisms, should be kept as simple and small as possible. The rationale behind this principle is that techniques such as code verification are required to discover design and implementation flaws which occur only under exceptional conditions. Such techniques are effective only if the design is simple and the actual code is minimized as much as possible.

1.2 FAIL-SAFE DEFAULTS

This principle asserts that access decisions should be based on permission rather than exclusion. This equates to the condition in which lack of access is the default, and the ``protection scheme'' recognizes permissible actions rather than prohibited actions. The authors mention that decisions based on exclusion present the wrong ``psychological base'' for a secure system. Also, failures due to flaws in exclusion-based systems tend to grant (unauthorized) permission, whereas permission-based systems tend to fail safe with permission denied.

1.3 COMPLETE MEDIATION

This principle stresses that ``every access request to every object must be checked for authority.'' This requirement forces a global perspective for access control, during all functional phases (e.g. normal operation, maintenance). Also stressed are reliable identification of access request sources and reliable maintenance of changes in authority.

1.4 OPEN DESIGN

This principle stresses that secrecy of design or the reliance on the ignorance of (malicious) users is not a sound basis for secure systems. Open design allows for open debate and inspection of the strengths, or origins of a lack of strength, of that particular design. Secrecy, a strong protection mechanism in itself, can be implemented through the use of passwords and keys. Secrecy implemented through the use of these objects is much easier to maintain than secrecy of design, where disclosure permanently effects system security. By comparison, passwords and keys are easily changed if disclosed, restoring security of the system.

1.5 SEPARATION OF PRIVILEGE

This principle asserts that protection mechanisms where two keys (held by different parties) are required for access are stronger mechanisms than those requiring only one key. The rationale behind this principle is that ``no single accident, deception, or breach of trust is sufficient'' to circumvent the mechanism. In computer systems the separation is often implemented as a requirement for multiple conditions to be met before access is allowed.

1.6 LEAST PRIVILEGE

This principle specifies that ``every program and user of the system should operate using the least set of privileges necessary to complete the job.'' One effect of this principle is that the potential for damage caused by an accident or an error is limited. This principle addresses the need for minimal interactions between privileged programs and the need to prevent improper uses of privilege. An example of the results of not applying the least privilege principle was seen in the November, 1988 Internet ``Virus'' [Eichin 1989]. One of the ways in which this virus gained access to its host systems was by exploiting the ability of the UNIX ``sendmail'' Simple Mail Transfer

Protocol (SMTP) server to execute arbitrary programs. The ability to execute arbitrary programs was not necessary to the operation of SMTP and, under the least privilege principle, should not have been provided. Its unnecessary presence gave an extra privilege, the execution of arbitrary programs on the remote host, to users of sendmail, and thus allowed the ``virus'' to exploit this privilege to compile and execute its initial components on remote systems it wished to ``infect.''

1.7 LEAST COMMON MECHANISM

This principle requires the minimal sharing of mechanisms either common to multiple users or ``depended upon by all users.'' Sharing represents possible communications paths between subjects which can be used to circumvent security policy. Global mechanisms must be adequately secure in terms of the user with the strictest requirements on the system; this encourages non-global mechanisms.

1.8 PSYCHOLOGICAL ACCEPTABILITY

This principle encourages the routine and correct use of protection mechanisms by making them easy to use, thus giving users no reason to attempt to circumvent them. Also addressed is the need for the supplied mechanisms to match the user's own image of his protection goals. This requirement prevents the user from attempting to implement his protection goals in an unnatural or arcane manner using poorly fitting mechanisms, since such an approach would be more prone to error.

2 ADDITIONAL DESIGN PRINCIPLES

Saltzer and Schroeder present the following principles as being employed in the area of physical security. They observe that these principles do not apply as well to computer systems as do the principles discussed earlier. We have found that this is similarly the case for some of the integrity mechanisms we have examined.

2.1 WORK FACTOR

This principle permits the use of mechanisms that are vulnerable to systematic attack, because the cost or amount of work involved in the attack is out of proportion to the value of the protected objects. It is a weaker mechanism by definition, since many alternate protection mechanisms are not vulnerable to systematic attacks, but can only be defeated by a hardware failure, design error, etc. Besides this inherent vulnerability, another problem is that the actual work factor which is implemented may be hard to determine exactly or may sometimes be reduced through automated attacks. In particular, a mechanism that may give an intolerable work factor in a manual system may be quite easy to attack on a computer system, since the computer, being designed to perform difficult, repetitive tasks, will perform the difficult work for the attacker. Thus the computer can often be used as a tool to assist in an attack upon its own security mechanisms.

2.2 COMPROMISE RECORDING

This principle calls for the accurate reporting of information compromise as a substitute for relying on more complex, preventive mechanisms. This principle is not as strong in computer systems, since the reporting mechanism itself must be protected from tampering and interference, otherwise violations may not be recorded. If the reporting mechanism is co-resident on the system being protected, an intruder may be able to gain access to the reporting mechanism at the same time the intruder gains access to other resources.

3 FUNCTIONAL CONTROL LEVELS

Saltzer and Schroeder also offer a categorization of protection functionality, or ``the kinds of access control that can be expressed naturally and enforced.'' This functionality is the result of design decisions, with higher levels of functionality being harder to attain in implementation. The authors also discuss a concept which is independent of the level of functionality: dynamics of use. This term refers to the way in which access privileges are specified, and how the system allows modification of that specification. They mention that while static expressions of protection are (relatively) easy to implement, dynamic specification (requests to change the specification which are made by executing programs) can introduce considerable complexity into the system.

Two problems associated with the dynamics of use are access determination and access revocation. The authors also mention that while the focus of their paper is on protection concerns which are internal to the computer, some functionality may be added through external protection methods. A short summary of the authors' categorization follows.

3.1 UNPROTECTED SYSTEMS

These systems are not stressed, but they are worth mentioning because they were the most common type then in use (1975). Essentially, these systems have no inherent, effective means of preventing any user from access to all information on the system. Examples are batch data processing systems and most popular PC operating systems. This is of concern in the light of the widespread use of PCs today, including analogous machines in the tactical systems environment. These systems offer ``mistake-prevention features,'' but no real security. Such features just ensure that breaches of control are intentional rather than accidental.

3.2 ALL-OR-NOTHING SYSTEMS

The main feature of these systems is the isolation of user environments and information. This isolation is sometimes moderated by such features as a public library mechanism, with users able to contribute information to that library. Still, the emphasis is on all users having equal access privileges, and all users have a view of the system as if it were a private machine.

3.3 CONTROLLED SHARING

A significant rise in complexity is required in these systems, which control access to each data item. Access control lists are a typical mechanism which enforces controlled sharing. Although conceptually simple, the control mechanisms tend to be intricate and difficult in implementation. Generic access types (e.g., read, write, execute) tend to be system defined to allow specification of user access privileges.

3.4 USER-PROGRAMMED SHARING CONTROLS

These systems allow users to restrict access to files in non-standard ways (as defined by the system). For example, time constraints may be imposed upon certain accesses, or multiple users' privileges (or actions) may be required for other accesses. These systems often implement the concept of protected objects and subsystems (emphasis in original), which effectively confines all accesses to the protected objects to programs within the protected subsystems. Entry to the subsystem is confined to specified entry points, allowing users to develop any programmable form of access control for objects created by the user. Other concepts include content-based and context-based (access) decision policies.

3.5 LABELLING INFORMATION

The previous three types of systems address the conditions under which an executing program is allowed access to information. In addition there are other systems which attempt to bring about continuing access control after the initial access decision. These systems also prevent or control the propagation of access privileges. A typical technique employed by these systems is the use of labels on data objects, which are used by the protection mechanism(s) to make access decisions. The authors state that such systems are rare and incomplete. This is still true today, but not to the same extent as when the paper was published.

ACRONYMS

ACF2	Access Control Facility 2
ACL	Access Control List
AIS	Automated Information Systems
C	Certification
CDI	Constrained Data Item
CIC	Combat Information Center
CPU	Central Processing Unit
DBMS	Database Management System
DDT	Domain Definition Table
DNA	Deoxyribonucleic Acid
DoD	Department of Defense
DTT	Domain Transition Table
E	Enforcement
I&A	Identification & Authentication
I/O	Input/Output
IC	Integrated Circuit
ICAP	Identity-based Capability
ID	Identification or Identifier
INFOSEC	Information Security
ISO	International Standards Organization
IVP	Integrity Verification Procedures
LED	Light Emitting Diode
LOCK	Logical Coprocessor Kernel
MMU	Memory Management Unit
NCSC	National Computer Security Center
PC	Personal Computer
RACF	Resource Access Control Facility
ROM	Read-Only Memory
SAT	Secure Ada Target
SCAP	Secure Capability
SMTP	Simple Mail Transfer Protocol
TCB	Trusted Computing Base
TCSEC	Trusted Computer System Evaluation Criteria

TOP	Tagged Object Processor
TP	Transformation Procedure
UDI	Unconstrained Data Item

GLOSSARY

abstract data type. A mechanism which associates a set of permissible values and operations with an identified class of objects. See **type.**

access control list. A special case of **access control tuples**, where the specification of access to an object is implied by grouping all subject and permitted operation information into a list, and attaching the list directly to the object.

access deterrence. A design principle for security mechanisms which is based on a user's fear of detection of violations of security policies rather than absolute prevention of violations.

access time minimization. A risk-reducing principle that attempts to avoid prolonging access time to specific data or to the system beyond what is needed to carry out requisite functionality.

access triple. A type of access control specification proposed in [Clark 1987] in which a user, program, and data item are listed for each allowed operation.

accountability. A principle which calls for holding individuals responsible for their actions. In automated systems, this is enabled through Identification and Authentication (I&A), the specification of authorized actions, and the auditing of the user's activity.

actor. A term used to describe an object in object-oriented systems, where the object consists of both data and the operations which manipulate the data contents.

agent. A term used to denote a user or an automated surrogate acting on behalf of a user.

alarm. A signal that warns or alerts [Webster 1988].

anticipation. A technique related to polling, in which the resource detects that it has not been accessed by an active subject within an expected interval.

atomicity. A property of a transaction in which each data item involved is either transformed from one unimpaired state to a new unimpaired state, or the initial unimpaired state is restored if the transaction fails.

authorization. The principle whereby allowable actions are distinguished from those which are not.

authorization override. The ability to take a definite action under exceptional circumstances to circumvent normal controls. authorized actions. The set of actions which a subject is allowed to perform.

availability. In computer security, availability denotes the goal of ensuring that information and information processing resources both remain readily accessible to their authorized users.

check digit. A checksum calculated on each digit of a numeric string (e.g., parity bits).

brevity codes. A shortened form of standardized messages which reduces the amount of input required by valid users while often hiding the message type, or other information obtained from the user interface, from unauthorized users.

capabilities. A protected identifier that both identifies the object and specifies the access rights to be allowed to the accessor who possesses the capability. In a capability-based system, access to protected objects such as files is granted if the would-be accessor possesses a capability for the object [NCSC 1988].

category. A restrictive label that has been applied to classified or unclassified data as a means of increasing the protection of the data and further restricting access to the data [NCSC 1988].

chained checksum. A checksum technique in which the hashing function is a function of data content and previous checksum values.

checksum. A numeric value which is computed (by some particular algorithm) based on the entire contents of a data object.

Chinese Wall model. A model presented in [Brewer 1989] to address confidentiality policies in the context of dynamically changing access permissions.

classification. (1) In data security, a determination that information requires, in the interest of national security, a specific degree of protection against unauthorized disclosure together with a designation signifying that such a determination has been made [Longley 1987]. (2) Systematic arrangement in groups or categories according to established criteria [Webster 1988].

conditional authorization. A type of authorization which is activated only at the occurrence of conditioning events.

conditional enabling. A type of control in which actions are physically disabled unless an authorization for the action is granted.

conditioning events. Events which are specified as being sufficient to invoke a reaction from a conditionally authorized subject.

confidentiality. The concept of holding sensitive data in confidence, limited to an appropriate set of individuals or organizations [NCSC 1988].

configuration management. The management of security features and assurances through control of changes made to a system's hardware, software, firmware, documentation, test, test fixtures and test documentation throughout the development and operational life of the system [NCSC 1988].

constraint. The state of being checked, restricted, or compelled to avoid or perform some action [Webster 1988].

cryptographic checksum. A checksum computed by an algorithm which produces a unique value for each possible data value of the object.

cryptography. The principles, means and methods for rendering information unintelligible, and for restoring encrypted information to intelligible form [NCSC 1988].

data attribute checks. A verification that data has particular attributes.

data compression. Techniques which reduce the size of data objects by encoding redundancies in the data to a more compact form.

data integrity. The attribute of data relating to the preservation of (1) its meaning and completeness, (2) the consistency of its representation(s), and (3) its correspondence to what it represents. See **integrity**.

data minimization. A generalization of the principle of **variable minimization**, in which the standardized parts of a message or data are replaced by a much shorter code, thereby reducing the risk of erroneous actions or improper use.

data movement primitives. A method of controlling shared objects by providing primitive operations which (logically) move the shared object into private addressing space when being accessed, thus preventing simultaneous access to the object by another process.

denial of service. Any action or series of actions that prevent any part of a system from functioning in accordance with its intended purpose. This includes any action that causes unauthorized destruction, modification, or delay of service [NCSC 1988].

digital signature. In authentication, a data block appended to a message, or a complete encrypted message, such that the recipient can authenticate the message contents and/or prove that it could only have originated with the purported sender. The digital signature is a function of (1) the message, transaction or document to be signed (2) secret information known only to the sender and (3) public information employed in the validation process [Longley 1987].

discrete value checks. A verification that data has either certain permissible values or does not have other restricted values, out of a wider set of possible values.

discretionary access control. A means of restricting access to objects based on the identity and need-to-know of the user, process and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject [NCSC 1988].

domain. The unique context (e.g., access control parameters) in which a program is operating. In effect, the set of objects that a subject has the ability to access [NCSC 1988].

duplication protocol. A communications protocol which duplicates the data being sent in order to provide fault tolerance.

duty. A required task, conduct, service, and/or function that constitute what one must do and the manner in which it should be done.

dynamics of use. Pertaining to the way in which access privileges are specified, and how the system allows modification of that specification [Saltzer 1975].

embedded system. A system that performs or controls a function, either in whole or in part, as an integral element of a larger system or subsystem [NCSC 1988].

encapsulation. The principle of structuring hardware and software components such that the interface between components is clean and well-defined, and that exposed means of input, output, and control other than those that exist in the interface do not exist.

encryption. See **cryptography**.

error correcting code. A technique in which the information content of the error-control data of a data unit can be used to correct errors in the message data of that unit.

error correction. Techniques which attempt to recover from detected data transmission errors.

event constraint. A type of **constraint** in which the active agent must perform an action or set of actions within a specified bound of time.

fault tolerance. The ability of a system to continue to perform its tasks after the occurrence of faults [Johnson 1989].

gate. An encapsulation implementation technique which provides access between domains only via specific locations (the gates) which provide a well-defined, controlled interface.

handshaking protocol. Techniques in which two or more communicating entities exchange information about successful or unsuccessful reception of data for error control.

hierarchical supervision. A technique in which integrity critical actions are handled or controlled by a more trusted or more experienced user while less critical actions are delegated to subordinates.

identity. The sameness in all that constitutes the objective reality of a thing: oneness; and is the distinguishing character of a thing: individuality [Webster 1988].

individuation. The determination of the individual in the general [Webster 1988].

integrity. (1) A subgoal of computer security which pertains to ensuring that data continues to be a proper representation of information, and that information processing resources continue to perform correct processing operations. (2) A subgoal of computer security which pertains to ensuring that information retains its original level of accuracy. (3) Sound, unimpaired or perfect condition [NCSC 1988]. See data integrity, system integrity.

label. Written or printed matter accompanying an article to furnish identification or other information [Webster 1988]. See **security label**.

least privilege. The principle that requires that each subject be granted the most restrictive set of privileges needed for the performance of authorized tasks. The application of this principle limits the damage that can result from accident, error, or unauthorized use [NCSC 1988].

mandatory access control. A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity [NCSC 1988].

minimization. A risk-reducing principle that supports integrity by containing the exposure of data or limiting opportunities to violate integrity.

minimizing target value. A risk-reducing practice which stresses the reduction of potential losses incurred due to a successful attack, and/or the reduction of benefits an attacker might receive in carrying out such an attack.

monitor. To watch, observe, or check-especially for a special purpose [Webster 1988].

mutual exclusion. The guarantee of exclusive access to a given data item for the purpose of completing a critical action in order to avoid interference by other entities attempting to simultaneously access the same data item.

N-person control. A method of controlling the actions of subjects by distributing a task among more than one (N) subjects.

noninterference. A concept proposed in [Goguen 1982] whereby a member of particular class cannot initiate actions that affect what is observed by members of a different class.

non-reversible action. A type of action which supports the principle of accountability by preventing the reversal and/or concealment of activity associated with sensitive objects.

notarization. The authentication of an object which is being transferred between two parties by an independent entity.

object. A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. Examples of objects are records, blocks, pages, segments, files, directories, directory trees, and programs, as well as bits, bytes, words, fields, processors, video displays, keyboards, clocks, printers, and network nodes [NCSC 1988].

obligation. The binding, constraining, or commitment of an individual or an active agent to a course of action.

privilege. (1) In operations, pertaining to a program or user and characterizing the type of operation that can be performed [Longley 1987]. (2) An **authorization** to perform an action.

privilege states. The states of a system in which a normally prohibited set of actions are allowed. Typically, use of privilege states is limited to ``trusted processes'' which are known or believed to use the normally prohibited actions only in specific, controlled ways. See **protection ring.**

process sequencing. A technique which controls the order in which specific tasks or subtasks are completed.

protection ring. One of a hierarchy of privileged modes of a system that gives certain access rights to user programs and processes authorized to operate in a given mode.

protocol. (1) A code prescribing strict adherence to correct procedures. (2) In telecommunications, it is a set of characters at the beginning and end of a message that enables communications between computers at various abstract service layers. These layers establish peer-entities between the communicating systems.

range checks. A verification that data is within a certain range of values.

read. A fundamental operation that results only in the flow of information from an object to a subject [NCSC 1988].

receive. A communications-oriented generalization of the **read** operation.

redundancy. (1) The part of a message that can be eliminated without loss of essential information. (2) The use of duplicate components to prevent failure of an entire system upon failure of a single component.

responsibility. Being answerable for what one does.

reversible action. An action that, once initiated, can be undone leaving the object being acted upon in a state as if the action had never been initiated.

risk reduction. The function of reducing one or more of the factors of risk, e.g., value at risk, vulnerability to attack, threat of attack, protection from risk.

role. A distinct set of operations required to perform some particular function.

rotation of duty. A method of reducing the risk associated with a subject performing a (sensitive) task by limiting the amount of time the subject is assigned to perform the task before being moved to a different task.

routine variation. The risk-reducing principle which underlies techniques which reduce the ability of potential attackers to anticipate scheduled events in order to minimize associated vulnerabilities.

run-to-run totals. Checksums which are applied to sequenced data, in which the hashing function is made a function of the previously seen data.

security label. A piece of information that represents the security level of an object [NCSC 1988].

security model. A formal presentation of the security policy enforced by the system. It must identify the set of rules and practices that regulate how a system manages, protects, and distributes sensitive information [NCSC 1988].

security policy. The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information [NCSC 1988].

send. A communications-oriented generalization of the **write** operation.

sensitive information. Any information, the loss, misuse, modification of, or unauthorized access to, could affect the national interest or the conduct of Federal programs, or the privacy to which individuals are entitled under Section 552a of Title 5, U. S. Code, but that has not been specifically authorized under criteria established by an Executive order or an act of Congress to be kept classified in the interest of national defense or foreign policy [NCSC 1988].

separation. An intervening space established by the act of setting or keeping apart.

separation of duty. The partitioning of tasks and associated privileges among different users or subjects, or to different, mutually-exclusive **roles** associated with a single user.

separation of name spaces. A technique of controlling access by precluding sharing-names given to objects are only meaningful to a single subject and thus cannot be addressed by other subjects.

serializability. Having the capability to consecutively order all actions in a logical transaction schedule.

strong typing. The application of **type enforcement** during all operations upon instances of **abstract data types** within a system or program.

summary integrity check. A method for checking the correct-

ness of data by comparing it with a ``summary'' of the data.

See **checksum.**

subject. An active entity, generally in the form of a person, process, or device, that causes information to flow among objects or changes the system state. Technically, a process/domain pair [NCSC 1988].

supervisory control. A method of controlling particular

actions of a subject by making those actions dependent on

the authorization of a supervisor.

system integrity. The attribute of a system relating to the successful and correct operation of computing resources. See **integrity.**

tactical system. A system which is used as an integral part of or in support of a weapon(s) system.

theft of service. The unauthorized use of information processing resources. time stamping. The method of including an unforgeable time stamp with object structures, used for a variety of reasons such as sequence numbering and expiration of data.

timeouts for inactivity. The setting of time limits for either specific activities or for non-activity. See anticipation.

Tranquillity Principle. A requirement that changes to an object's access control attributes be prohibited as long as any subject has access to the object.

transmittal list. A list, stored and transmitted with particular data items, which identifies the data in that batch and can be used to verify that no data are missing.

Trojan horse. A computer program with an apparently or actually useful function that contains additional (hidden) functions that surreptitiously exploit the legitimate authorizations of the invoking process to the detriment of security or integrity [NCSC 1988].

trusted computing base (TCB). The totality of protection mechanisms within a computer system, including hardware, firmware, and software, the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of a TCB to enforce correctly a unified security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g., a user's clearance level) related to the security policy [NCSC 1988].

trusted path. A mechanism by which a person at a terminal can communicate directly with the TCB [**trusted computing**

base]. This mechanism can only be activated by the person or the TCB and cannot be imitated by untrusted software [NCSC 1988].

two-phase commit protocol. A mechanism which requires the logging of a record for all actions to ensure that any transaction either completely succeeds, or may be rolled back so that effectively no changes have occurred.

type. In programming, pertaining to the range of values and valid operations associated with a variable [Longley 1987].

type enforcement. The enforcement of allowable operations on an instance of an **abstract data type**, and on the permissible values that instance may take. See **strong typing.**

unconditional authorization. A type of authorization which enables immediate action on the part of the authorized subject.

value checks. A precursor to type enforcement mechanisms, value checks are checks against a definition of meaningful values through explicit enumeration and/or exclusion.

variable minimization. A method of reducing the number of variables to which a subject has access to the minimum required, thereby reducing the risk of malicious or erroneous actions by that subject. This concept can be generalized to include **data minimization**.

version control. A mechanism which allows distinct versions of an object to be identified and associated with independent attributes in a well-defined manner.
virus. A self-propagating Trojan horse, composed of a mission component, a trigger component, and a self-propagating component [NCSC 1988].

well-formed transaction. Proposed in [Clark 1987] as a mechanism to prevent a user from manipulating a data item arbitrarily, but rather in constrained ways that preserve or ensure the internal consistency of data.

wholeness. Having all its parts or components-includes both the sense of unimpaired condition (i.e., soundness) and being complete and undivided (i.e., completeness).

write. A fundamental operation that results only in the flow of information from a subject to an object [NCSC 1988].

X.25. An International Organization for Standards (ISO) telecommunications standard for the network and data link levels of a communications network.