NCSC-TG-022
Library No. 5-236,061
Version 1

FOREWORD

A Guide to Understanding Trusted Recovery in Trusted Systems provides a set of good practices related to trusted recovery. We have written this guideline to help the vendor and evaluator community understand the requirements for trusted recovery, as well as the level of detail required for trusted recovery at all applicable classes, as described in the *Department of Defense Trusted Computer Systems Evaluation Criteria.* In an effort to provide guidance, we make recommendations in this technical guideline that are not requirements in the Criteria.

The *Trusted Recovery Guide* is the latest in a series of technical guidelines published by the National Computer Security Center. These publications provide insight to the *Trusted Computer Systems Evaluation Criteria* requirements for the computer security vendor and technical evaluator. The goal of the Technical Guideline Program is to discuss each feature of the *Criteria* in detail and to provide the proper interpretations with specific guidance.

The National Computer Security Center has established an aggressive program to study and implement computer security technology. Our goal is to encourage the widespread availability of trusted computer products for use by any organization desiring better protection of its important data. One way we do this is by the Trusted Product Evaluation Program. This program focuses on the security features of commercially produced and supported computer systems. We evaluate the protection capabilities against the established criteria presented in the *Trusted Computer System Evaluation Criteria.* This program, and an open and cooperative business relationship with the computer and telecommunications industries, will result in the fulfillment of our country's information systems security requirements. We resolve to meet the challenge of identifying trusted computer products suitable for use in processing information that needs protection.

invite your suggestions for revising this technical guideline. We will review this document as the need arises.

30 December 1991

Patrick R. Gallagher, Jr.
Director
National Computer Security Center

ACKNOWLEDGMENTS

The National Computer Security Center extends special recognition and acknowledgment to Dr. Virgil D. Gligor as the primary author of this document. James N. Menendez and Capt. James A. Muysenberg (USAF) are recognized for the development of this guideline, and Capt. Muysenberg is recognized for its editing and publication.

We wish to thank the many members of the computer security community who enthusiastically gave their time and technical expertise in reviewing this guideline and providing valuable comments and suggestions.

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 BACKGROUND

The principal goal of the National Computer Security Center (NCSC) is to encourage the widespread availability of trusted computer systems. In support of this goal the NCSC created a metric, the *DoD Trusted Computer System Evaluation Criteria (TCSEC)* [17], against which computer systems could be evaluated.

The *TCSEC* was originally published on 15 August 1983 as CSC-STD-001-83. In December 1985 the Department of Defense adopted it, with a few changes, as a Department of Defense Standard, DoD 5200.28-STD. DoD Directive 5200.28, *Security Requirements for Automatic Information Systems (AISs)* [10], requires the Department of Defense to use the *TCSEC*. The *TCSEC* is the standard used for evaluating the effectiveness of security controls built into DoD AISs.

The *TCSEC* is divided into four divisions: D, C, B, and A. These divisions are ordered in a hierarchical manner. The *TCSEC* reserves the highest division (A) for systems providing the best available level of assurance. Within divisions C and B are subdivisions known as classes, which also are ordered in a hierarchical manner to represent different levels of security in these divisions.

1.2 PURPOSE

An important assurance requirement of the *TCSEC,* which appears in classes B3 to A1, is *trusted recovery*. The objective of *trusted recovery* is to ensure the maintenance of the security and accountability properties of a system in the face of failures and discontinuities of operation. To accomplish this, a system should incorporate a set of mechanisms enabling it to remain in a secure state whenever a well-defined set of anticipated failures or discontinuities occur. It also should include a set of procedures enabling the administrators to bring the system to a secure state whenever unanticipated failures or discontinuities occur. (Chapter 6 explains the distinction between anticipated and unanticipated failures.)

Besides these mechanisms, the *TCSEC's* B3-A1 classes require the implementor to follow specific design principles and practices, collectively called *assurance* measures. The *TCSEC* further requires the developer to provide specific *documentation evidence* sufficient for an evaluator or accreditor to verify that the mechanisms and assurances are sufficient to meet specified requirements.

This guide presents the issues involved in the design of trusted recovery. It provides guidance to manufacturers on what functions of trusted recovery to incorporate into their systems. It also provides guidance to system evaluators and accreditors on how to evaluate the design and implementation of trusted recovery functions. This document contains suggestions and recommendations derived from *TCSEC* objectives but which the *TCSEC* does not require. Examples in this document are not the only way of accomplishing trusted recovery. Nor are the recommendations supplementary requirements to the *TCSEC*. The only measure of *TCSEC* compliance is the *TCSEC* itself.

This guideline isn't a tutorial introduction to the topic of recovery. Instead, it's a summary of trusted recovery issues that should be addressed by operating systems designed to satisfy the requirements of the B3 and A1 classes. We assume the reader of this document is an operating system designer or evaluator who is already familiar with the notion of recovery in operating systems. The guide explains the security properties of system recovery (and the notion of trusted recovery). It also defines a set of baseline requirements and recommendations for the design and evaluation of trusted recovery mechanisms and assurance. The reader who is unfamiliar with the notion of system recovery and security modeling required of B3 and Al systems may find it useful to refer both to the recovery literature (such as [1, 5, 14-16, 20-23, 25, 27]) and the security literature (such as [3,11, 26, 29]) cited in this guide.

1.3 SCOPE

Trusted recovery refers to mechanisms and procedures necessary to ensure that failures and discontinuities of operation don't compromise a system's secure operation. The guidelines for trusted recovery presented refer to the design of these mechanisms and procedures required for the classes B3 and A1 of the *TCSEC*. These guidelines apply to computer systems and products built or modified with the intention of satisfying *TCSEC* requirements. We make additional recommendations derived from the stated objectives of the *TCSEC*.

Not addressed are recovery measures designed to tolerate failures caused by physical attacks on ADP equipment, natural disasters, water or fire damage, nor administrative measures that deal with such events. The evaluation of these measures is beyond the scope of the *TCSEC* [17, p. 89].

1.4 CONTROL OBJECTIVE

Trusted recovery is one of the areas of operational assurance. The assurance control objective states:

"Systems that are used to process or handle classified or other sensitive information must be designed to guarantee correct and accurate interpretation of the security policy and must not distort the intent of that policy. Assurance must be provided that correct implementation and operation of the policy exists throughout the system's life-cycle." [17, p. 63]

This objective affects trusted recovery in two important ways. First, the design and implementation of the recovery mechanisms and procedures must satisfy the life-cycle assurance requirements of correct implementation and operation. Second, both a system's administrative procedures and recovery mechanisms should ensure correct enforcement of the system security policy in the face of system failures and discontinuities of operation. The notions of failure and discontinuity of operation are defined in Chapter 2.

1.5 DOCUMENT OVERVIEW

This guide contains five chapters besides this introductory chapter. Chapter 2 reviews the key notions of failure, discontinuity of operation, and recovery. Chapter 3 discusses the properties of trusted recovery. Chapter 4 presents recovery design approaches and options that can be used for trusted recovery. Chapter 5 discusses the impact of the other *TCSEC* requirements on trusted recovery. Chapter 6 presents *TCSEC* requirements that affect the design and implementation of trusted recovery functions, and includes additional recommendations corresponding to B3-A1 evaluation classes. The glossary contains the definitions of the significant terms used. Following this is a list of the references cited in the text.

2.0 FAILURES, DISCONTINUITIES, AND RECOVERY

The *TCSEC* requires for security classes B3 and A1 that:

"Procedures and/or mechanisms shall be provided to assure that, alter an ADP system failure or other discontinuity, recovery without a protection compromise is obtained." [17, p. 39]

In this chapter we discuss the notions of failure and discontinuity of Trusted Computing Base (TCB) operations, and present an informal qualitative description of their effects on system states. We also briefly present general recovery approaches used in practice. Throughout this chapter and document we use the term "failure" for an event causing a system function to behave inconsistently with its informal specification. We reserve the term "discontinuity" of operation for failures caused by user, administrator, or operator action.

Recovery mechanisms of computer systems are designed to respond to anticipated failures or discontinuities of operation. These mechanisms do not handle "unanticipated" failures nor "unanticipated" discontinuities of operation; therefore, computer-system documentation should include descriptions of administrative procedures to handle such events. In a well-designed system, unanticipated failures and discontinuities of operation are events expected to occur with very low frequency, i.e., once or twice per year. For this reason, administrative procedures, as opposed to automated mechanisms in the system, represent an adequate response to unanticipated failures and discontinuities of operation, even when these procedures are complex and extensive.

One can't establish formal models of failure and discontinuity of operation in which proofs demonstrate the model's internal consistency. Neither physical systems, such as devices, processing units, and storage, nor behaviors of users, administrators, and operators, have formal properties [21]. Therefore, formal modeling and specification of expected failures and discontinuities of operation can't be required. Only informal assumptions derived from operational experience can be made about expected failures, discontinuities, their effects, and their frequencies. References (14, 15, 21] present examples of such assumptions. These informal assumptions, which should be stated explicitly in system documentation, form the basis for the design of the recovery mechanisms and the definition of the administrative recovery procedures.

However, recovery mechanisms and administrative procedures must reconstruct consistent system states, or prevent state transitions to inconsistent states, as a direct response to occurrences of expected failures or discontinuities of operation [8, 9]. A system state is "consistent" if the variables defining it satisfy given predicates expressing formally or informally invariant properties of the system, discussed in Section 3.1. A "state transition" is a function which changes the variables of a system state in a specified way, i.e., specified as constraints on the system's rules of operation—discussed in Section 3.2. Therefore, the design of recovery mechanisms and administrative procedures should use invariant properties and state-transition constraints of the security model defined for the system, viz., discussion in Chapter 3.

The role of recovery mechanisms and of trusted recovery can be best understood by illustrating the effect of failures and discontinuities of operation on typical systems. Informal and qualitative assumptions of failures derived from operational experience with various systems have been presented in the literature [14,15, 21]. Using these informal assumptions we can define general classes of failures that affect the operation of a TCB.

One class of failures is identical to the class of errors caused when users pass wrong parameters to TCB primitives, or invoke the wrong TCB primitives, and when system resources are exhausted or found in an inconsistent state because of user actions. These are called state-transition failures or action failures. We cover this type of user-induced failure, which falls more naturally in the area of exception processing, for two reasons: (1) the failures of this class are, nevertheless, TCB domain failures regardless of their cause; and (2) the processing of these failures—not just their specification and documentation—is relevant to system security.

For example, incorrect error processing can bring the system into a state where a user cannot communicate with the TCB, or can contribute to the mishandling of covert channels. However, we place the major emphasis in this guideline on the more traditional notions of failure, namely TCB failures, media failures, and administrator-induced discontinuity of operation.

2.1 STATE-TRANSITION (ACTION) FAILURES

State-transition failures, also called action failures, occur whenever a TCB primitive, which causes a state transition, cannot complete its function because it detects exceptional conditions during its execution. State-transition failures can be caused by bad parameters passed to TCB primitives, by exhaustion of resource limits, by missing objects needed during TCB primitive execution, and so on.

The effects of state-transition failures on TCB states are not as far-reaching as those of other failures. Because these failures occur often, the code of TCB primitives usually includes recovery mechanisms that undo the temporary modifications of system states before the primitive's return, thus returning the system to a consistent state. If the recovery mechanisms of TCB primitives fail to undo temporary modifications of system states, the system may remain in an inconsistent state and eventually crash. A crash is a failure that causes the processors' registers to be reset to some standard values [21]. Because consistent system states cannot be recovered from processor and primary memory registers after a crash, these registers are referred to as "volatile" storage. In contrast, consistent system states can usually be recovered from magnetic media such as disks and tapes; these media are called "nonvolatile" storage.

Examples of recovery mechanisms included in TCB primitives to undo temporary state modifications after state-transition failures are found in most contemporary operating systems. For instance, consider the "creat" primitive of a hypothetical UNIX(R) system which allocates i-node table entries before allocating file table entries [1]. If the file table entry is full at the time "creat" call is made, a state-transition failure would occur. Before returning to the caller, the recovery code of "creat" deallocates the i-node table entry allocated for the file that couldn't be created. Failure to deallocate such entries would cause the i-node table to fill up and remain full, causing a system crash.

(R) UNIX is a registered trademark of UNIX System Laboratories, Inc

2.2 TCB FAILURES

TCB failures occur whenever the TCB code detects an error below the TCB primitives' interface which can't be fixed; i.e., the error cannot be masked. TCB failures are caused by persistent inconsistencies in critical system tables, by wild branches of the TCB code (possibly caused by transient hardware failures), by power failures, by processor failures, and so on. TCB failures always cause a system crash.

In systems providing a high degree of hardware fault tolerance, system crashes still occur because of software errors. Since crashes cause volatile storage to be lost, and since nonvolatile media usually survive crashes, recovery mechanisms can reconstruct consistent states in a maintenance mode of operation. After reconstructing a consistent state, the recovery mechanisms restart the system with no process execution in progress, e.g., processes that were active, blocked, or swapped out before the crash are aborted. New processes, which run the code of aborted processes executing at the time of the crash, can be started by users after the consistent state is reconstructed. Recovery mechanisms can reconstruct consistent states by either removing or completing incomplete updates of various objects represented on nonvolatile media. Properties of and design approaches for recovery mechanisms able to reconstruct consistent states from nonvolatile storage after TCB failures are discussed in Section 3.2 and Chapter 4.

Some TCB failures allow a system to shut down in an orderly manner. These failures may be caused by process swap-space exhaustion, timer-interrupt table exhaustion, and, in general, by conditions that can't be handled by TCB primitives themselves in normal modes of operation. Traps originated by persistent hardware failures, such as memory and bus parity errors, also may cause failures.

2.3 MEDIA FAILURES

Media failures occur whenever errors are detected on some nonvolatile storage device that the TCB cannot fix (i.e., the errors can't be masked). Media failures are caused by hardware failures such as disk head crashes, persistent read/write failures due to misaligned heads, worn-out magnetic coating, dust on the disk surface, and so on. They also are caused by software failures such as TCB failures which make media unreadable.

The effect of media failures is that part, or all, of the media representing TCB objects become inaccessible and corrupt. Data structures relevant to system security also may be corrupted by media failures, e.g., object security labels. The system usually crashes unless the lost data can be retrieved from archival storage and rebuilt on a redundant storage device. Of course, media failures that don't affect TCB objects may not cause system crashes. If redundant media aren't available, or if users and administrators don't keep archival data up-to-date, media failures may become unrecoverable failures. Administrative recovery procedures may have to be used to bring the system to a consistent state. As discussed in Chapters 5 and 6, all these procedures should be explained in the system's Trusted Facility Manual.

2.4 DISCONTINUITY OF OPERATION

Failures induced by users, administrators, and operators cause discontinuities of operation. Inside an operating system, discontinuities of operation manifest themselves most often as state-transition failures, TCB failures, and, less often, as media failures. They are caused by erroneous actions, such as unexpected system shutdowns, e.g., by turning off the power. Also, they can be caused by lack of action, such as ignoring the exhaustion of critical system resources under administrative control despite documented or on-line warnings, e.g., audit trail is 95% full, insufficient swap space left, inadequate configuration installed, etc.

The effects of discontinuities of operation are the same as those of the state-transition and TCB failures mentioned above. Recovery mechanisms or administrative procedures necessary for the reconstruction of a consistent state also are correspondingly similar to those used for failures. For example, cancellation of a TCB primitive call by depressing the “break” key during the call's execution might have the same effect as a state-transition failure detected by the TCB primitive. Each TCB primitive and state transition would have to be designed either to ignore user cancellation signals during execution of critical code sections or to clean up internal data structures during the processing of such signals.

Actions such as system shutdowns by power-off action during execution of TCB code may cause TCB failures. Recovery mechanisms for TCB failures caused by power failures also may be able to handle unexpected system shutdowns. In either case, during subsequent power-on procedures, the TCB not only detects that TCB failures left the system in an inconsistent state, but also initiates recovery of a consistent state before the system enters the normal mode of operation.

Somewhat less often, administrator or operator actions cause media failures. For example, initiation of on-line diagnostic tests of a media controller during normal mode of system operation, instead of the maintenance mode, would most likely cause media failures. Similarly, initiation of TCB maintenance actions such as disk reformatting in the normal mode of operation would certainly cause subsequent media failures. Discontinuity of operation caused by administrator- or operator-induced failures may require use of administrative recovery procedures.

3.0 PROPERTIES OF TRUSTED RECOVERY

The properties of trusted recovery are defined in terms of two notions: *secure states* and *secure state transitions*. A system state is secure whenever consistency invariants derived from valid interpretations of security and accountability models are satisfied. A state transition is secure if both its input state and its output state are secure, and it satisfies the constraints placed on it by valid interpretations of security policy and accountability policy models.

Accountability models include models of user authentication, trusted path, and audit. The notions of invariants for secure states and constraints for specific state transitions are briefly illustrated in this chapter and discussed in detail in reference [11]. Reference [29] discusses the notion of a valid interpretation of a security model in detail and reference [3] illustrates it. For the sake of brevity, interpretations of security models aren't illustrated in this guideline.

3.1 SECURE STATES

State-machine (or "state-transition") models of security, such as the Bell-La Padula model [3], define a state in terms of the following abstract variables:

a. subjects (trusted and untrusted)

b. objects

c. access privileges

d. access matrix (defining the privileges of subjects to objects)

e. current access set (defining the privileges subjects can currently exercise over objects)

f. security function (defining the subject's maximum and current subject clearance and the object classification)

g. object hierarchy

These abstract variables are used in defining state invariants that help define the notion of the secure state. The following paragraphs labeled (1)-(5) discuss the use and characteristics of state invariants for trusted recovery.

(1) Security invariants are derived formally from security model interpretations.

State-machine models also include conditions, or axioms, whose interpretations in a given system provide *invariant properties* which must be satisfied by secure states. For example, the conditions of the Bell-La Padula model include the following:

a. the simple-security condition for subjects

b. the *-property for the security function

c. the discretionary security condition for access privileges of current access sets

d. the compatibility condition for object hierarchies

The model description [3] precisely defines the specific meaning of these conditions in terms of the variables of the system states and the examples of their valid interpretations.

(2) Informally derived invariants should augment formally derived invariants whenever necessary.

State-transition models might not include all the conditions that are relevant to the notion of security or accountability. Whenever this is true, new invariants need to be defined to augment the set of existing invariants derived from interpretation of model conditions (or axioms). For example, additional invariants may be defined for objects such as the password file, user-account file, security map file, and system configuration file, which are used by trusted processes of a system supporting the Bell-La Padula model. These invariants may refer to multiple types of objects, as illustrated in this example:

In all system states, all user and group identifiers must be unique, integer values; the identifiers' length may vary from zero to a defined maximum number of characters.

User and group identifiers may be included in a password file, a user account file, and a file defining group membership. Additional invariants also may be needed for areas of security and accountability policy where the interpretations of the model's conditions provide insufficient detail for a given system. For example, an invariant that is specific to TCBs implementing Multics-like access control lists (ACLs) [26] for discretionary access control may state:

In all system states, the entries of an ACL must be sorted as follows:

- <user id.group id > entries precede <user id.* > entries
- <user id.* > entries precede < *.group id > entries
- < *.group id: > entries precede < *.* > entries

where "*" represents the wild-card qualifier.

Similar invariants also should augment other areas of access control and accountability (e.g., invariants for user and system security profiles including minimum and maximum user clearances and object classifications; invariants for audit mechanisms).

Recovery mechanisms of a TCB become trusted only if they maintain secure states in the normal mode of operation, or detect insecure states and reconstruct secure states in the maintenance mode of operation, despite the occurrence of failures and discontinuities. To detect insecure states after system failures or to verify that recovered states are secure, recovery mechanisms must check whether security invariants are satisfied. All security invariants are relevant to the recovery mechanisms handling state-transition failures. These failures usually leave the system in the normal mode of operation, instead of causing the system to enter maintenance mode. Therefore, all invariants which must hold in the normal mode of operation also must hold after recovery from state-transition failures.

(3) Some security invariants may be irrelevant for trusted recovery.

Not all security invariants are relevant for other classes of failures. For example, consider the case of TCB failures when all volatile memory is lost and the system enters maintenance mode. In maintenance mode, consistent states are recovered, security invariants are checked, and the system is placed back in normal mode of operation in a secure state that usually includes an empty set of user processes (i.e., untrusted subjects) and a corresponding empty current access set.

In most system-recovery designs, recovery mechanisms restart only a few of the trusted processes (e.g., the "login" process listening to terminals). User processes in execution at the time of TCB failure are aborted because their state is lost either along with the loss of volatile memory or during the recovery of secure states. Users can start new processes executing the code of these aborted processes after the system is placed in normal mode of operation. Therefore, after TCB failures, recovery mechanisms need not check invariant properties referring to the current access set.

For example, from the set of the invariants derived from the conditions of the Bell-La Padula model, only those corresponding to the simple-security condition and from the compatibility condition remain relevant for trusted processes and for the object hierarchy, respectively, after system crashes. However, if the current state of at least some user processes can be recovered after a TCB failure, other invariants derived from the conditions of the Bell-La Padula model (e.g., the *-property), also become relevant. Thus, the types of relevant invariants depend on the type of desired and possible system-state recovery (e.g., a state with no active processes), no opened files or devices, or a state with at least some active processes, opened files and devices. The type of system-state recovery is determined by either the designers' choice or by other, non-*TCSEC* requirements. However, a complete set of security invariants should be derived from interpretations of security models for any type of system-state recovery.

(4) All TCB integrity invariants should be satisfied by trusted recovery.

To detect insecure states, or to verify that recovered states are secure, recovery mechanisms also must verify whether integrity invariants of the TCB are satisfied. Internal TCB integrity invariants do not necessarily refer only to state variables defined by user-level security or integrity models. They also may refer to internal TCB variables used for object and subject representation, and may not be visible either at the user nor at the administrative interface of the TCB. For example, internal TCB-integrity invariants may require the satisfaction of the conditions below.

For all recovered states:

a. A disk sector is either free or allocated.

b. The sum of free and allocated disk sectors equals the total number of disk sectors.

c. Every allocated disk sector occurs exactly once in exactly one object representation, and all free disk sectors do not belong to any object.

d. All active objects are reachable from the root of their hierarchy.

e. If an object's link count or reference count is zero, neither a link nor a reference exists to that object.

Most operating systems provide recovery mechanisms which attempt to detect internal inconsistencies after crashes and to recover consistent states. [1, §5.18] provides examples of internal operating system invariants for the UNIX file system which are enforced by the file system checking program "fsck."

(5) Lack of security invariants makes trusted recovery impossible.

The key role of secure-state determination in trusted recovery underlines the importance of security invariants. Lack of such invariants makes it impossible to determine whether TCB primitives can tolerate state-transition failures, i.e., whether TCB primitives clean up temporary state modifications before they return to their caller. Similarly, lack of security invariants make it impossible for the recovery mechanisms and procedures to determine whether the recovered state is secure even if all state transitions are designed to satisfy all model's constraints in the face of "expected" failures and discontinuities of operations, discussed in Section 3.2. "Unexpected" TCB failures, media failures, or discontinuities may still leave the system in insecure states. Whether such states are secure or insecure can only be determined by verifying the state invariants.

3.2 SECURE STATE TRANSITIONS

A secure state transition originating from a secure input state guarantees that (1) its output state is secure, and that (2) *all security constraints* defined for it are satisfied. In contrast with an invariant, which is defined on the variables of individual states and holds for all state transitions, a constraint is a predicate that relates the variables of two or more states. A constraint may be relevant only to specific state transitions. A typical constraint is the following:

For any TCB primitive which changes the security level of an object, the new level must dominate the original security level of the object.

This constraint expresses a policy requirement that users can only upgrade, but not downgrade, the classification of an object. Other, similar constraints can be found in the appendix of reference [3], and in reference [11].

(1) Trusted recovery requires the satisfaction of all constraints.

TCB primitives implement secure state transitions. The primitives may temporarily place the system in insecure states before they return to their caller. Any failure or discontinuity of operation during a state transition may leave the system in an insecure state. Furthermore, even if the recovered system state appears secure, it's possible that the constraints placed on the secure state transition (during which the TCB failed) are violated as a result of the failure.

For example, consider a state transition which changes the security level of an object and satisfies the constraint defined above in absence of failures. Because a security level may be a multi-field, multi-word data structure, any update to it requires the execution of several instructions including at least one I/O instruction. A failure may occur in the middle of the security level update; e.g., a power failure during the disk-write operation may leave the security level only partially updated. This may violate the constraint defined above but may not affect any state invariant. The update may have changed the clearance field and some, but not all, of the category fields of the security level before the failure. As a consequence, the resulting object security level would no longer dominate the original object security level, violating the above constraint. However, it's possible the resulting object security level satisfies all state invariants, such as those derived from the interpretation of the simple-security and compatibility conditions of the Bell-La P'adula model. Recovery mechanisms would have no way to detect whether the recovered state, which would include the update, is insecure.

Similar, and more probable, problems may appear when the TCB crashes in the middle of writing a label of a newly created file, in the middle of a security map update by a system-administrator process, or in the middle of an update to a file containing an access control list or any other security-relevant data structure. The notion of atomicity, discussed in the paragraph below labeled (2), would prevent this type of recovery problem from violating system-security policy.

Recovery mechanisms can detect that the system is in an insecure state following a TCB or media failure, after extensive, time-consuming checks of security invariants, *without* relying on special designs of TCB code implementing secure state transitions. However, seldom are the recovery mechanisms able to recover an old secure state, or construct the new secure state from nonvolatile media, *without* relying on special design and implementation of TCB code that anticipates various types of failures.

Because constraints relate variables of old states with those of the new states created by secure state transitions, *failures may make it impossible to verify constraints.* In other words, the values of the old states may no longer be available, whereas the values of the new states may not be fully updated at the time of the failure. Thus, it's important to design the TCB code implementing state transitions so assurances provide the recovery of either the secure input state or the secure output state of a transition. In the latter case, the recovery mechanism should be guaranteed to satisfy the transition's security constraints—not just the secure-state invariants.

(2) Atomicity of all TCB primitives ensures trusted recovery.

Atomicity is the key property of the TCB code implementing secure state transitions, enabling the recovery mechanisms to reconstruct secure states despite the occurrence of (anticipated) failures. Assuming that all state transitions can be serialized by concurrency control mechanisms [21], a state transition is "atomic" in the face of failures if it's either carried out completely or has no effect at all. In practice, this property implies that the TCB code implementing state transitions is designed to ensure that, for the class of anticipated failures, the recovery mechanisms are always able to reconstruct either input secure states or output secure states after failures.

The literature describes various approaches for providing atomicity at the operating system level [16, 21, 23, 25, 27], and at the data base management system level [14,15]. We summarize these approaches in Chapter 4.

(3) Atomicity of all TCB primitives is not always necessary for trusted recovery.

In many operating systems or TCB primitives, it may be difficult to ensure that *all* secure-state transitions are atomic. Some operating system primitives consist of many individual object updates, and atomicity requires the implementation of these primitives as "transactions" (defined in Section 4.2.2 and in references [14, 15, 16, 21, 23, 25, 27]). The performance penalties and complexity of implementing transactional behavior in all TCB primitives might be prohibitive for many small operating systems.

In many small operating systems, a more practical approach is taken to the design of TCB primitives and object representations to aid recovery mechanisms in the detection of inconsistencies. It's acceptable to implement TCB-primitive mechanisms which only help detect inconsistent (insecure) states after crashes, but do not ensure atomicity of secure-state transitions. These implementations leave the task of reconstructing consistent or secure states to administrative users and tools. The choice of which TCB primitives should be made atomic (if any) or which TCB primitives should only help detect insecure states after crashes (but not be atomic) belongs to the system designers.

An example of a file system design allowing the detection of inconsistent states by a "scavenging" process is provided in [22]. The scavenging process recovers files whose representation is intact, and deletes files whose representations become inconsistent after crashes. To a large extent, the UNIX system calls are designed to ensure that the "fsck" program can detect a variety of file system inconsistencies after crashes, with only some facilities for consistent state recovery without administrative intervention, viz., [1, Chs. 5.16-5.18] and [5]. Section 4.2 presents examples of design considerations for TCB primitives which help the recovery mechanisms detect and correct inconsistencies.

4.0 DESIGN APPROACHES FOR TRUSTED RECOVERY

In this chapter we discuss the responsibilities of administrative users in trusted recovery, outline some practical difficulties in designing trusted recovery mechanisms, and summarize several nonexclusive design approaches to recovery that may be used in trusted systems. We also review the main options for trusted recovery available to operating system designers. The primary goal of this chapter is to give the reader an overview of the recovery design issues presented in the literature and implemented in various systems during the past decade.

4.1 RESPONSIBILITY FOR TRUSTED RECOVERY

The responsibilities for trusted recovery fall into two categories depending on the type of failure that occurred: (1) TCB design responsibility, and (2) administrator responsibility. State-transition failures are routinely handled by designs of TCB primitives without placing the system in maintenance mode. Controlled system shutdown and subsequent rebooting also is performed by the TCB with little or no administrative intervention, viz., the “fsck” program of UNIX.

In contrast, emergency system restarts and cold starts are performed primarily by system administrators using TCB-supported tools. Among the system administrators, the System Programmer role is responsible for trusted recovery functions (e.g., consistency checks on system objects, on individual TCB files and databases, repair of damaged security labels, and so on [24]). The Security Administrator also may perform recovery responsibilities whenever the System Programmer’s role is not separately identified by the system’s design.

The System Programmer role may have additional tools necessary for checking the integrity of TCB structures and the security invariants of the initial state, i.e., for establishing the initial secure-system state. The determination of this state is similar to the determination of whether a recovered state is secure. However, the initial system state may not necessarily be identical to a recovered state because, unlike a recovered state, the initial state may not contain any user-visible objects. Thus, fewer invariants may be relevant for determining the security of the initial states.

Tools such as the ones suggested in reference [2], for identifying vulnerabilities of certain system states, also may be useful both for establishing secure initial states and recovered states. Because some of the tools used for detecting insecure states and for repairing various system data structures place the TCB in maintenance mode (e.g., fsck tools [1, 5]), the use of these tools should be restricted to the System Programmer role.

4.2 SOME PRACTICAL DIFFICULTIES WITH CURRENT FORMALISMS

The key role of secure state invariants and of state-transition constraints in trusted recovery suggests that designs of B3-A1 systems should use state-machine models of security policy [11], instead of other types of models, such as information flow models, which are not defined in terms of secure states and secure-state transitions. Although in principle this suggestion appears to be sound, in practice the existing state-machine models of security policy, such as [3], are not fully adequate for designing trusted recovery mechanisms. This is the case for at least the following two reasons.

First, when current state-transition models are interpreted, the state invariants and transition constraints for trusted processes acting on behalf of system administrators are inadequate in number. Second, extant models do not include accountability properties of secure systems, thus making the formal derivation of accountability invariants and constraints impossible. The apparent reason for these inadequacies of extant state-machine models is that they are too abstract to include system-specific invariants and constraints for the areas mentioned above.

Invariants and constraints that need to be defined for trusted processes and for administrative programs in the accountability area are significantly more numerous and complex than those derived from extant state-machine models for other TCB areas. An attempt to define state invariants in the trusted process area —for formal security-verification purposes, not for trusted recovery—is reported in [4]. Appendix E of reference [19] defines a fairly extensive list of invariant conditions for state variables in the accountability area for the specific purpose of verifying the security of initial and recovered system states.

Lack of *formal* invariants and constraints for state variables and state transitions of trusted processes and accountability programs makes it difficult to determine formally whether TCB primitives can tolerate even state-transition failures, e.g., determine formally whether inconsistent states are cleaned-up properly before primitives return to callers. To determine formally whether recovery mechanisms can reconstruct current states, or complete state transitions in maintenance mode after expected TCB or media failures, is a challenging task for anyone. However, *informal* derivation of system-specific invariants and constraints is acceptable for design of trusted-recovery mechanisms so long as evidence is provided of their correct and extensive use in design of these mechanisms. Such use represents significant added assurance these mechanisms are designed correctly.

4.3 SUMMARY OF CURRENT APPROACHES TO RECOVERY

4.3.1 Types of System Recovery

Operating systems' responses to failures can be classified into three general categories: (1) system reboot, (2) emergency system restart, and (3) system cold start [14].

System reboot is performed after shutting down the system in a controlled manner in response to a TCB failure. For example, when the TCB detects the exhaustion of space in some of its critical tables, or finds inconsistent object data structures, it closes all objects, aborts all active user processes, and restarts with no user process in execution. Before restart, however, the recovery mechanisms make a best effort to correct the source of inconsistency. Occasionally, the mere termination of all processes frees up some important resources, allowing restart with enough resources available. Note that system rebooting is useful when the recovery mechanisms can determine that TCB and user data structures affecting system security and integrity are, in fact, in a consistent state.

Emergency system restart is done after a system fails in an uncontrolled manner in response to a TCB or media failure. In such cases, TCB and user objects on nonvolatile storage belonging to processes active at the time of TCB or media failure may be left in an inconsistent state. The system enters maintenance mode, recovery is performed automatically, and the system restarts with no user processes in progress after bringing up the system in a consistent state.

System cold start takes place when unexpected TCB or media failures take place and the recovery procedures cannot bring the system to a consistent state. TCB and user objects may remain in an inconsistent state following attempts to recover automatically. Intervention of administrative personnel is now required to bring the system to a consistent state from maintenance mode.

4.3.2 Current Approaches

A possible view of automated recovery mechanisms for TCBs would be to consider all internal TCB data structures and security attributes as a database. This view is useful because significant technological advances in database consistency and recovery have been made in the last decade; in principle, one could use these advances for the design of trusted recovery for TCBs. For this reason, we review possible approaches used in data management and other storage systems for implementing atomic actions and transactions.

An alternative approach to recovery, used mostly at the operating system level relies on detection of inconsistencies caused by failures during non-atomic state transitions and subsequent correction of those inconsistencies. This approach, generally called the "optimistic" approach to recovery (and to the serializability of actions), assumes that TCB failures are rare, and therefore the overhead penalties of its extensive recovery mechanisms don't affect the overall performance significantly. In general, the optimistic approach to recovery has better overall performance than approaches which make all state transitions atomic, but recovery is significantly more difficult to design. Section 4.3.4 provides examples of operating system mechanisms that help detect TCB inconsistencies after failures.

4.3.3 Implementation of Atomic State Transitions

An action is atomic if it's *unitary* and *serializable*; it's unitary if it either happens or has no effect; it's serializable if it's part of a collection of actions where any two or more actions relating to the same object appear to execute in seemingly serial order [14, 15, 21, 25, 27]. A transaction consists of a sequence of read and write actions, and is atomic if its entire sequence of actions is unitary and serializable. Examples of TCB primitives which should be implemented as atomic actions or transactions include updating a password file, updating a security map, changing a security level, changing a user security profile, updating an ACL, and linking or unlinking an object to a hierarchy.

Three basic techniques are used to implement atomic actions and transactions: (1) update shadowing, (2) update logging, and (3) combinations of update shadowing and logging. A major difference between shadowing and logging is that systems using shadowing accomplish all updates to redundant disk pages or files until all updates of an atomic transaction are finished, and then introduce these updates into the original system data; whereas systems using only logging write each update on a log first and then update the original system data directly, i.e., the updates are done in place.

The advantages and disadvantages of both alternatives, the storage and 110 requirements for nonvolatile memory, and the notions of two-phase commits in transaction implementations are discussed in [14,15].

4.3.3.1 Shadowing

The central idea behind making a TCB primitive behave as an atomic transaction is that of updating the primitive in two phases. First, all the updates of the primitive are collected in a set of "intentions," without changing the system's data, i.e., the updates are not performed in place. The last action of the primitive is to commit its updates by adding a special "commit" record to the set of intentions.

Second, the updates of the intentions set replace the original data in the system, making the updates visible to other system actions or transactions.

If a crash occurs after the commit record is written, but before all intentions replace the original data on nonvolatile storage, the second phase is restarted by the recovery mechanisms as often as necessary (because there may be subsequent crashes during recovery), without leaving any undesirable side effects. Thus, the recovery mechanisms can complete a state transition interrupted by the crash by carrying out all update intentions. Then the set of intentions is erased.

The recovery mechanisms are able to construct the new secure (output) state which would have been produced by the state transition had failure not interrupted it. If a crash occurs before writing the commit record, the recovery mechanisms can only find, or reconstruct, an incomplete set of intentions on nonvolatile storage and erase them. Thus, the recovery mechanisms retain the original secure (input) state prevailing before the crash interrupted the state transition.

To ensure that the above scheme works correctly for an expected set of TCB and media failures, the nonvolatile media must be designed in such a way that (1) the recovery mechanisms can find or reconstruct complete sets of intentions after a crash or determine the intentions list is incomplete (e.g., no commit record is found for it); and (2) the action which writes the commit record, and therefore completes the set of intentions, is itself atomic.

To ensure property (1) above, redundant storage is used to represent sets of intentions, e.g., a pair of related disk pages are written sequentially with the same intention data. The placement of each page of a pair on nonvolatile media, which defines the pair's relationship, is chosen so at least one of the two pages survives all "expected" failures. For example, if the redundant storage containing intentions lists is expected to survive disk-head crashes, the disk should be duplexed and page "*a*" on the first disk also is written as page "*a*" on the second disk. However if the redundant storage is only expected to survive disk-read failures, page "*a*" also is written as page "*f(a)*" on the same disk, where the function "*f*" defines the relationship between the addresses of the two pages. The only inconsistency-detection and reconstruction tasks to be performed relate to the recovery of complete intentions sets after each crash. Note that the inconsistency-detection and reconstruction tasks are placed at a low level within the TCB and, therefore, need not concern themselves with the maintenance of type properties of various objects the TCB may have updated at the time of the crash.

To ensure (2) above, the last disk-write operation to the intentions set, which commits the transaction, should be implemented with a single hardware-provided 110 instruction that's either atomic or detectably non-atomic with respect to the set of expected failures. Most commercially available I/O hardware satisfies this requirement.

This scheme, called *shadowing*, was conceived by Lampson and Sturgis in 1976, was reported in detail in [21], and was implemented in several distributed storage systems at Xerox PARC [20, 23, 25, 27].

4.3.3.2 Logging

Mechanisms which assure that a TCB primitive exhibits transaction atomicity and which are based on logging assume that nonvolatile storage exists which can be made reliable enough to survive all expected failures, e.g., by storage duplexing. With logging, each update of a TCB primitive, including both the original object values and the update values, is written onto a log represented on nonvolatile storage before the object being updated is actually modified in storage, i.e., updated in place. The last action of the primitive is to write a commit record on the log, signifying the end of the primitive's invocation.

If the system crashes before the commit point, the entire TCB primitive, i.e., transaction, is aborted. Because the log is written before any individual update is made in place, all updates can be undone during recovery merely by reading the log records of the primitive and restoring the original values saved in the log entries. Complex consistency checks are unnecessary. Thus, the recovery mechanisms can retain the original (input) state prevailing before the state transition was interrupted by the crash. This protocol is called "the write-ahead log" protocol in [14, 15] and implemented in [18]. However, if the system crashes after writing the commit record onto the log, then all updates of that TCB primitive can be redone from the log records. Thus, the only reconstruction activity remaining is that of restoring object contents from the log records.

Logging mechanisms similar to the one just outlined have been used in database management systems for a long time. Complete description of similar logging mechanisms, which in practice also may include state checkpointing and transaction savepointing features, are found in [14, 15], and in systems reference manuals of many commercially available database management systems [18].

4.3.3.3 Logging and Shadowing

Both logging and shadowing have relative advantages and disadvantages. These are discussed in [15]. Gray et al. point out that the performance characteristics of shadowing make it a desirable mechanism for small databases and systems, whereas logging is desirable in large database systems. In practice many systems combine both mechanisms to retain the advantages of each [15, 16].

In general, operating systems and TCBs maintain relatively small databases for implementing security policies. Many of the extant operating systems, such as UNIX, already use simple versions of file shadowing to do some atomic actions and transactions. Few operating systems implement logging at the TCB level for recovery reasons. Logging is used mostly at the application level, outside the TCB.

Both informal [21] and formal [16] proofs of correctness for recovery mechanisms using shadowing, or shadowing and logging, are in the literature. All these proofs make only informal assumptions of failure behavior.

4.3.4 Recovery with Non-Atomic State Transitions

Many of the extant operating systems take an "optimistic" approach to recovery. Designs of such systems assume failures requiring system restart are rare. Therefore, the performance penalties in normal mode of operation and design complexity caused by TCB primitives with atomic behavior may be unwarranted. Whether such penalties are significant is still a matter of some debate. For some performance figures the reader should consult references [14] and [23).

Subtle system-specific properties are encountered in designs of TCB primitives that don't support atomic state transitions but nevertheless help ensure that the recovery mechanisms can reconstruct a consistent system state. In this section, we present an example of a generic source of inconsistency caused by TCB crashes, and illustrate specific properties of non-atomic TCB primitives enabling recovery of consistent, secure states. Of course, these properties aren't necessarily sufficient for trusted recovery. Although other similar properties may be found for trusted recovery, demonstrating their sufficiency for trusted recovery with non-atomic TCB primitives remains a challenge of individual system design.

4.3.4.1 Sources of Inconsistency--A Generic Example

A typical inconsistency caused by TCB crashes appears in operating systems maintaining several related data structures within the TCB to enable the sharing, protection, and management of objects. For example, user-level references to an object, which are stored in directories, typically point to a single "object map" (e.g., an entry in the "object map table," in the "master object directory," in the "global object table," in a "volume table of contents," or an i-node in the "i-node space"). This map contains various fields defining object attributes such as length, status (active/inactive, locked/unlocked, access and modification time, etc.), access privileges or ACL references, and a list of memory blocks allocated to the object referenced by the map. The map identifies the object's representation on secondary storage. A copy of all the object maps representing active objects is kept in primary memory. These copies identify the representation of objects in a cache of active objects. The memory blocks allocated to the object representation may themselves contain user- level references to other objects, e.g., whenever the object is a directory.

Any TCB primitive which deletes such an object has to invalidate and/or deallocate at least two of the three related data structures, viz., the object representation and the object map. In most systems, the outstanding user-level references to the deallocated object, e.g., user-directory entries for the deallocated object, also have to be invalidated or deleted. Capability-based systems are an exception to this because capabilities are user-level object references which can't be reused owing to their use of system-wide unique identifiers [12].

The invalidation or deletion of the three types of related structures (*user-level references* => *object map* => *object representation*) during a TCB-primitive invocation should be performed in a single atomic action or transaction. Similarly, any TCB-primitive invocation which creates an object and a reference to it would have to allocate all three related structures in a single atomic action or transaction. Should the TCB primitives allocate/deallocate the three related structures independently using non-atomic actions, crashes between the allocation/deallocation of one of the related structures and the rest might leave references pointing to nonexistent objects, causing a "dangling-reference" problem. Alternatively, such crashes might leave references to already allocated objects of other users, causing an "object-reuse" problem.

4.3.4.2 Non-Atomic TCB Primitives

Some designs of TCB primitives enable recovery mechanisms to reconstruct consistent states without requiring atomicity of (all) TCB primitives and (all) object updates. We present two properties of TCB designs below illustrating this fact.

Example 1-- Ordered Disk-Writes within TCB Primitives

TCB primitives which allocate/deallocate TCB structures, such as the three related structures mentioned previously, could order their write (update) operations to nonvolatile storage in such a way as to prevent both dangling references and object reuse problems after crashes. Consider the following order of updates and requirement for synchronous writes:

> (1) In any TCB primitive which deallocates user-level references, object map, and object representations within one invocation, the necessary disk- write operations should follow the direction of references; i.e., the objects containing the user-level reference should be written to the disk first, the object map next, and the object representation last.
>
> In any TCB primitive which allocates user-level references, object map, and object representation within one invocation, the necessary disk operations should follow the direction opposite to that of references; i.e., the object representation should be written to the disk first, the object map next, and the object containing the user-level reference last.
>
> (2) All disk-write operations of a TCB primitive which updates user-level references, object map, and object representation within one invocation, should be synchronous; i.e., the caller should not be allowed to proceed until the disk-write operation completes.

The usefulness of ordering disk-writes in all relevant TCB primitives is apparent when one considers the effects of system crashes and subsequent TCB rebooting. Suppose the order of disk-write operations mentioned above does not hold for a TCB primitive which deallocates an object along with the last reference to it. In this scenario, the object map and representation could be deallocated first, and their corresponding tables could be updated by two disk-write operations before the object containing the user-level reference would be updated.

A crash occurring after the completion of the first two disk-write operations leaves the user-level reference dangling. This requires the recovery procedures to search the entire directory system to find the dangling reference (if any) before the TCB is rebooted and before the deallocated object map and representation space are reallocated. Should this not happen, or should recovery procedures fail to discover dangling references after crashes, the system might enter an insecure state causing object-reuse problems. A similar problem appears when the crash occurs after the deallocation of the object representation but before the deletion of the object map and the user-level reference.

In contrast, if the disk-write operations required by object deallocation and reference deletion are ordered as suggested in requirement (1) of Example 1, a crash between two consecutive disk-write operations could only cause objects or their maps to become inaccessible. This might cause a denial-of-service problem but not an object-reuse problem.

For example, a crash could occur after the last user-level reference was deleted, and after the corresponding disk-write was completed, but before the disk writes required by the object map and representation deletions could be completed. Both the object representation and map become inaccessible to any user-level programs.

Object inaccessibility also could be caused by crashes occurring after both user-level references and object map are deleted. However, to handle object inaccessibility, recovery procedures need only do “garbage collection” operations. Should these operations (inherently simpler than those finding all dangling references) not be done or should they fail to find all inaccessible objects after a crash, denial-of-service (but not security policy violations) may occur. This example shows that the proper ordering of disk-write operations in all relevant TCB primitives helps reduce the security vulnerability of a system after crashes.

The requirement (2) above for synchronous disk-write operations is necessary to preserve the ordering of disk-writes suggested in requirement (1). Asynchronous disk-write operations don'' necessarily preserve such ordering.

Similar disk-write ordering rules apply to multi-leaf trees of references. Such a tree appears in sets of related data structures (*user-level references* => *object map* => *object representation*) when a reference to a separate ACL object is placed in, or is implicitly associated with, the object map (i.e., *object map* => *object representation* and *object map* => *ACL*). In such a case, both the object representation and the ACL should be created before the object map, and the object map should be deleted before both the object representation and the ACL. Failure to order the disk-write operations in the proper sequence may cause the reuse of the ACL with foreign objects after crashes. This would represent a serious access control problem if left uncorrected.

Example 2—Redundancy of TCB Storage Structures

The idea of maintaining redundant storage for critical data structures, enabling recovery mechanisms to reconstruct consistent system states, is neither new nor novel. Similar ideas have led to the use of “double-entry bookkeeping” in accounting systems to accomplish similar goals. Furthermore, it could be argued the design approaches for atomic actions or transactions of TCB primitives—as discussed in the previous section—also use redundant storage structures. We suggest here that redundant storage structures can be used solely to aid recovery mechanisms to detect, and possibly correct, inconsistent TCB structures, and not necessarily to provide failure atomicity for all TCB primitives.

Although minimized TCBs, as those required for security classes B3 and A1, are likely to present at their interface only objects with a very simple structure, e.g., segments as opposed to files, the representation of these objects on nonvolatile storage usually requires the maintenance of other more primitive structures. The discovery of inconsistent or even inaccessible objects after a crash, but not necessarily the provision of atomic object updates, can be assured by maintenance of redundant, low-level TCB structures.

For example, consider a system in which a segment representation on the disk consists of a set of not necessarily contiguous pages (i.e., disk sectors) identified by an index page. To enable recovery mechanisms to detect inconsistent segment structures, redundant structures can be maintained independently of index pages that define the unique association between object identifiers and pages. All TCB primitives would be designed to reflect every update of an index page in the redundant structure also, and vice versa; however, updates would be reflected only after disk writes to originals had completed successfully (i.e., disk writes to index pages are synchronous). Recovery mechanisms would detect, and possibly correct, inconsistencies of segment representation by comparing the contents of index pages with those of the corresponding redundant structures. The additional ability to detect which of the two structures is affected by a failure, and the ability to ensure that one of the two structures survives all expected failures, enables recovery mechanisms to correct either one of the two structures, as necessary.

Redundancy of segment representation structures can't guarantee that the segment contents would be internally consistent, nor that they would be consistent with the contents of other segments. Failure-atomicity of TCB primitives wouldn't be guaranteed in any way by the maintenance of redundant structures suggested in Example 2, although similar structures could be used as the basis for additional mechanisms which would implement failure atomicity [23].

4.3.4.3 Idempotency of Recovery Procedures

Recovery procedures should be restartable after repeated crashes, and every time they're restarted, the TCB should be left in no worse state than before. This property is called *idempotency*, and should exist regardless of the type of disk-writes or redundancy used within the TCB. Repeated crashes of recovery procedures would have no undesirable side effects in the system's TCB. Although recovery procedures of systems supporting atomic TCB primitives and transactions also must be idempotent, the demonstration of idempotency of these procedures seems significantly simpler than in systems in which failure-atomicity of TCB primitives isn't supported. The idempotency property of recovery mechanisms using "intention sets"—discussed in Section 4.3.3.1—is presented in [21].

4.3.4.4 Recovery With Non-Atomic System Primitives

Example 1 — The UNIX File System

The ordering of synchronous disk-write operations is enforced by UNIX kernels [1]. For example, whenever the "unlink" primitive deletes the last link to an object and deallocates the object, and whenever the "creat" primitive allocates an object and places a reference to it in a directory, the disk-write operations are ordered as suggested in (1) of Section 4.3.4.2 and are performed synchronously. User-level references are represented in UNIX by i-node numbers in directory entries, object maps by i-nodes, and object representations by disk blocks. Reference [1, ch. 5] describes the negative consequences of not ordering the disk-write operations, and of not doing them synchronously.

The UNIX program "fsck" is also a good example of a typical recovery program. Fsck detects dangling i-nodes and i-node numbers, duplicate i-nodes associated with the same disk block, unbalanced i-node references and object links, as well as lost and corrupted i-nodes and disk blocks. However, for dangling i-nodes and i-node numbers, duplicate i-nodes associated with the same disk block, and unbalanced i-node references and object links, "fsck" requires the intervention of administrative users to resolve inconsistencies. As pointed out in [5], seldom is enough information available enabling administrators to make correct decisions and resolve inconsistencies. Suggested changes proposed in [5] would help administrators make sound low-level recovery decisions. Thus, not only should non-atomic system primitives be designed to support recovery but recovery mechanisms should enable administrators to make correct recovery decisions.

Example 2—The Cambridge File Server

The Cambridge File Server (CFS) actually implements failure-atomic updates of individual files [23]. Although a good example of disk redundancy, the mechanism provided by CFS can be used to guarantee only that recovery mechanisms can reconstruct consistent file structures, not necessarily file contents.

In the CFS, a file's representation on the disk consists of one or more index pages pointing to data pages. A file can be viewed as a single-root tree whose nodes are index pages and whose leaves are data pages. CFS maintains a redundant structure, called the cylinder map, to represent the relationship between file identifiers, file pages, and page status for all objects of each disk cylinder. Each cylinder map entry contains (1) the unique identifier of the file to which the defined page belongs, (2) the tree address occupied by the page in the file representation, and (3) the allocation and use state of the page. Thus, each disk page is defined by its entry in the cylinder map and by its position in the tree of pages representing the structure.

The CFS primitives update the cylinder maps only after a file's tree is updated consistently. Thus, the CFS recovery procedures can rebuild the file system structure using redundant information after a crash-producing failure. For example, whenever a crash corrupts a file's index page, the recovery procedures search the cylinder maps to find all pages belonging to that file. At the end of the search, all pages referenced by the corrupted index page and their tree positions are found, and the corrupted index page can be reconstructed. Conversely, whenever a cylinder map is corrupted by a crash, the recovery procedures examine each file structure starting at the root of the file system to find all disk pages belonging to the corrupted cylinder map. At the end of the search, which may include the entire file system and may take as long as half an hour with current disk technology, the corrupted cylinder map is reconstructed.

The cylinder maps of the CFS are used for additional functions, including that of keeping status information for pages of intention sets. Although the use of the redundancy function of cylinder maps in providing recoverable storage for implementing atomic actions has been less successful than that used in shadow paging [23], the use of this function for automated recovery of file system structures is quite adequate.

Example 3—Selective Atomicity

Another approach to recovery in a secure state without using atomic TCB primitives is "selective atomicity." Selective atomicity means that only a subset of the TCB actions are atomic when operating on a specific subset of TCB objects. The intent is to preserve the consistency of the representation of certain objects (e.g., those implemented in nonvolatile storage) and object hierarchies (e.g., file systems) but not necessarily of object contents. Whenever it's security-relevant, the consistency of object contents is left in the care of the particular application. Special atomicity mechanisms maintain the consistency of the content of security-relevant objects—a small subset of all system's objects.

The advantages of using the selective-atomicity approach are that the performance penalties of full atomicity are avoided and system recovery in a secure state is simplified. For example, recovery programs could restore the structure of a file system (including directories, disk blocks, and indices to physical records), minimizing administrative intervention—a feature not found in most recovery programs such as the "fsck."

Selective atomicity exists in the AIX system version 3.1. In this system, a fairly robust file system is obtained by implementing atomic updates for segments that contain directories, i-nodes, and indirect file blocks. Also, any change to the diskblock allocation map is atomic [6]. The implementation of the atomicity features is based on logging using a concept called "database memory," because of its resemblance to the logging features of database management systems [7]. Note that the selective atomicity implemented in AIX 3.1 does not refer to non-kernel objects (e.g., objects implemented by system processes). Thus, a TCB using the AIX 3.1 kernel would have to carry out atomic or recoverable actions within trusted processes whenever such actions are required.

4.4 DESIGN OPTIONS FOR TRUSTED RECOVERY

The design and implementation of trusted recovery mechanisms and prOcedures should include the following nonexclusive options:

(1) For the set of all state-transition failures, TCB recovery code should ensure the restoration of the secure input state; i.e., remove all temporary modifications of the secure state which violate security invariants.

(2) For the set of expected TCB or media failures (appropriately chosen by the system designers), TCB code that causes state transitions should be designed, whenever possible, to make secure-state transitions atomic; i.e., the recovery mechanisms should reconstruct either the secure input states or the secure output states of those transitions.

(3) For the subset of state transitions that aren't atomic in the face of expected TCB or media failures, the recovery mechanisms should detect that these state transitions have left the TCB in insecure states. TCB-supported administrative tools should enable the reconstruction of a predictable secure state, with or without administrative-user intervention. This state may differ from both the secure input and output states of the transition during which the failure occurred.

(4) For the complementary set of "unexpected" or rare TCB and media failures, administrative procedures and tools (used to restore a predictable secure state) should be defined and documented. This secure state also may differ from both the secure input and output states of the transition during which the failure occurred.

Options (3) and (4) above suggest that expected TCB or media failures caused by either spontaneous events or user-, administrator-, or operator-induced discontinuities may create secure states which, nevertheless, violate integrity and availability requirements of user applications. Clearly, if the recovered state differs from either the secure input or secure output states of the transition during which the failure occurred, then both "lost" updates and "dirty" reads are possible [13,14]. However, options (3) and (4) are still acceptable for systems evaluated under the *TCSEC* because the *TCSEC* does not include application integrity and availability requirements.

If user applications require higher degrees of integrity and availability than those supported by the TCB, they could always implement additional, separate recovery mechanisms. In fact, this approach is taken by most current applications including database management systems [14, 15]. This approach is also sound from a system architecture point of view because it separates application recovery mechanisms from those of the TCB. Support of application level recovery within the TCB is both unnecessary and unwarranted. It's unnecessary because application portability rules out reliance on the recovery features of a specific TCB, and thus applications tend to implement their own recovery features. It's unwarranted because it violates the class B3-A1 requirement for TCB minimality and increases the assurance burden.

Trusted recovery requires that the state-transition models used for secure system design include *both* state invariants and transition constraints. A model that includes only state invariants is inadequate because trusted recovery requires that state-transition constraints be satisfied (viz., discussion in Section 3.2). Similarly, a model that includes only transition constraints is inadequate because the occurrence of unanticipated failures and discontinuities of operations may prevent the system from completing state transitions and place it in insecure states. These states can only be determined to be insecure after checking that secure-state invariants are not satisfied (viz., discussion in Section 3.1).

The reader should note that state-transition models of security policy are particularly suitable for the design of trusted recovery mechanisms. Because these models include the notion of secure states and secure state transitions, they can be integrated with recovery models, all defined in terms of states and state transitions (which are possibly different). Unlike the state-transition models, information flow and noninterference models don't include explicitly the notions of state and state transitions, and thus are more difficult, if not impossible, to use for defining formally the notion of trusted recovery. Furthermore, information flow and noninterference models cover nondiscretionary access controls and thus lack discretionary access control and other policy components. This makes the use of such models impractical for the formal definition of trusted recovery.

5.0 IMPACT OF OTHER *TCSEC* REQUIREMENTS ON TRUSTED RECOVERY

Security policy and accountability requirements of the *TCSEC* are only indirectly relevant to trusted recovery. That is, specific requirements of these areas, which may be relevant to trusted recovery, have already been levied on trusted facility management functions and interfaces [24]. In this chapter, we focus only on the *TCSEC* areas specific to trusted recovery; we discuss the relevance or irrelevance of specific requirements.

5.1 OPERATIONAL ASSURANCE

Most of the assurance requirements of the *TCSEC* apply to trusted recovery because trusted-recovery code is part of a system's TCB. Some *TCSEC* assurance requirements become irrelevant because interfaces to trusted recovery functions are either invisible to users or, whenever they are visible, can be used only by administrative personnel authorized by trusted facility management. The user visibility of trusted recovery interfaces, or lack thereof, is established under the assurance requirements of trusted facility management [24], and therefore we don't repeat it here.

In the operational assurance area, only the trusted facility management and the system architecture areas have specific requirements relevant to trusted recovery. Because system integrity requirements refer to the diagnostic testing of the hardware and firmware elements of the TCB, they have no special relevance here beyond that of addressing hardware/firmware elements that may include recovery mechanisms. Covert channel analysis of TCIB interfaces offered by trusted recovery isn't necessary.

Administrative users are the only users who may use trusted recovery mechanisms. They have multilevel access to system and user data and are trusted to maintain the data secrecy and not exploit covert channels while operating in administrative roles. Thus, administrative users must be cleared to the highest level of data classification present on the system. Furthermore, all code implementing trusted recovery functions should be scrutinized to ensure, to the largest extent possible, these functions don't contain any Trojan Horses or Trap Doors.

Most system-architecture requirements of the TCB apply to trusted-recovery code. For example, TCB programs and data structures implementing trusted recovery must comply with the following requirements:

a. Satisfy modularity requirements.

b. Make significant use of abstraction and information hiding.

c. Use layering of recovery functions.

d. Satisfy the requirements of the least privilege principle to the largest possible extent.

Since trusted recovery is used mostly in maintenance mode when all storage, segmented or not, must be available to recovery code, most protection mechanisms are disabled. Thus, application of the least privilege principle and insistence on use of logically distinct objects with separate attributes is less obvious here than when mechanisms are used in the normal mode of system operation.

The only trusted facility management requirement affecting trusted recovery is that of segregating the security-relevant from the security-irrelevant administrative functions. Because trusted recovery functions are obviously security relevant, they must be allocated either to the System Administrator or to the System Programmer roles [24].

5.2 LIFE-CYCLE ASSURANCE

In contrast with the operational assurance, all areas of life-cycle assurance are relevant to trusted recovery. These areas are security testing, design specification and verification, configuration management, and trusted distribution.

5.2.1 Security Testing

The purpose of testing trusted recovery mechanisms is to uncover design and implementation flaws allowing failure recovery to place the TCB in insecure states. The major issue in this area is delimiting the scope of security testing, i.e., reconciling the general objectives and practices of security testing with the limited coverage of failures and discontinuities of operation which is possible in practice.

The objectives of security testing suggest that security testing should be performed using test fixtures external to the TCB, should not require TCB instrumentation, should be repeatable, and should include precise coverage analysis. For testing trusted recovery functions, only the requirements of class B3 are relevant because, as discussed below, formal top-level specifications aren't necessary for trusted recovery, viz.,[24].

However, only state-transition failures and discontinuities of operation (but not all TCB and media failures) can be generated from outside the TCB in a repeatable manner and without any TCB instrumentation. Whenever TCB and media failures cannot be generated from outside the TCB in a repeatable manner and without any internal TCB instrumentation, design and implementation analysis and review are necessary to determine whether the recovery mechanisms can handle the untested failure responses. State-transition failures can be generated using similar test-plan structures and programs as those used for security testing of other TCB areas, e.g., testing security policy enforcement and covert channel bandwidth.

Discontinuities of operation can be generated using administrative interfaces in ways causing at least some TCB, and possibly media, failures repeatedly. In contrast, spontaneous TCB and media failures cannot be regenerated without software and hardware instrumentation of the TCB. Use of such instrumentation would violate one of the major objectives of security testing. Recall that TCB instrumentation is undesirable because either it precludes testing the system in normal mode configuration or it leaves test fixtures that may become exploitable Trap Doors in the TCB. Therefore, security testing of trusted recovery functions is limited to the use of test plans, i.e., test conditions, data, and coverage analysis, which cover only state-transition failures and discontinuity of operation, as defined in Chapter 2. Within this limited context, all conventional security-testing requirements and recommendations are applicable.

5.2.2 Design Specification and Verification

Inherent inability to define formal models of TCB failures and discontinuities of operations (viz., Chapter 2 of this guideline and reference [21]), and general lack of formal models of trusted facility management and administrative roles, makes the *TCSEC* requirement for top-level specification correspondence with the formal policy model irrelevant to trusted recovery. However, the requirement for use of a security policy model, and more precisely for a state-machine model, is relevant to trusted recovery in two areas.

First, state-machine models enable designers and implementors to define the notions of secure system states and state transitions. These notions are the key to trusted recovery as they provide the security invariants and constraints recovery mechanisms should satisfy. Recovery functions earn their trust only if they satisfy these invariants and constraints, as discussed in Chapter 4.

Second, the response of TCB primitives to state-transition failures (defined in Chapter 2) is modeled by formal security-policy models and specified by top-level specifications. For example, the Bell-La Padula model represents a clear, albeit incomplete, attempt to model these failures through the provision of the "error" and ''?'' elements of the TCB response set (D_m) to invocations of TCB requests (R_k) [3]. Thus, the specification of error messages and exceptions provided by TCB primitives in response to state-transition failures also is required for trusted recovery reasons.

5.2.3 Configuration Management

All configuration management requirements of classes B3 and A1 apply as stated.

5.2.4 Trusted Distribution

All trusted distribution requirements of class A1 apply to the TCB functions and interfaces implementing trusted recovery as stated.

5.3 DOCUMENTATION

Most documentation requirements of the classes B3 and A1 apply to trusted recovery as stated in each evaluation class. However, some requirements, such as those stating the need for a Security Features Users' Guide (SFUG) and for covert channel documentation, are obviously not relevant. The SFUG is relevant for nonadministrative users whereas trusted recovery is exclusively a responsibility of system administrators. The administrators are implicitly trusted not to disclose classified and proprietary information they can obtain from the system directly Without having to use covert channels.

5.3.1 Trusted Facility Manual

The Trusted Facility Manual (TFM) requirements are not only relevant but important to trusted recovery. The TFM must include the description of procedures necessary "to resume secure system operation after any lapse of system operation." Thus the TFM should include a description of the types of TCB failures and discontinuities of operation and a list of procedures, tools, warnings, and examples of how these failures might be best handled.

All TCB recovery procedures must be defined in the TFM. These procedures include analyzing system "dumps" after crashes, crash-recovery and restart actions, checking the consistency of TCB files and directories, changing system configuration parameters (e.g., table sizes, devices and device drivers, etc.), running periodic system-integrity checks, and repairing object inconsistencies and damaged labels. A list of the approved tools for TCB recovery, relevant commands, exceptions, warnings, and advice also should be in the TFM.

5.3.2 Test Documentation

The trusted recovery testing documentation consists of test plan, test program, and test result documentation. The general structure of the trusted-recovery test plan and test results is the same as that of all other test plans and results. For example, the test plans should contain a test condition section, a test data section (i.e., including a test environment setup, test parameters, and expected test outcomes), and test coverage analysis.

However, the content of these sections should differ substantially from that of other test plans. For example, the test conditions should identify the type of discontinuity of operation (and the induced TCB or media failure) generated by using the administrative interfaces for the current test. In the test data area, the environment setup should define the system initialization data, including TCB and user-level data structures and objects, which are necessary to generate the specified discontinuity of operation. The parameters and the commands used by administrators to generate discontinuity of operation also should be listed.

The outcomes of the test should include the specification of the automated (e.g., reboot, warm-start) procedures and of the manual (e.g., cold-start, emergency restart) procedures for trusted recovery and their expected effects on the system. The coverage analysis should explain the scope of the tests in terms of the classes of discontinuities covered by the test and the classes of spontaneous failures remaining uncovered because of inability to induce them by administrative action.

5.3.3 Design Documentation

The documentation of the trusted-recovery design should include the following items corresponding to the B3 and A1 requirements of the *TCSEC*:

a. Description of the anticipated classes of failures and discontinuities of operation handled, automatically or using administrative procedures, by trusted recovery.

b. Trusted recovery philosophy (e.g., use of failure-atomicity in the design of TCB primitives, of non-atomic actions which allow recovery of secure states, the type of recovered secure states—input-secure state, output-secure state of a transition, or some arbitrary secure state).

c. Warnings about the "unanticipated" failures that can't be handled in a routine manner.

d. State-security invariants and constraints maintained by trusted recovery.

e. Descriptive Top-Level Specification (DTLS) of the TCB primitives implementing trusted-recovery functions.

The accuracy of the design documentation should be commensurate with that of other similar documentation for B3 and A1 systems. In this area there are no substantive differences between the B3 and A1 requirements. This is true for the same reasons as those discussed in the design specification and verification area.

6.0 SATISFYING THE *TCSEC* REQUIREMENTS

In the *TCSEC*, there are no requirements for Trusted Recovery for security classes below class B3. Furthermore, security policy and accountability requirements which also may apply to trusted recovery are already included in the requirements for trusted facility management [24]. This chapter includes only additional requirements and recommendations specific to trusted recovery.

6.1 REQUIREMENTS FOR SECURITY CLASS B3

6.1.1 Operational Assurance

6.1.1.1 System Architecture

The TCB programs and data structures implementing trusted recovery must meet the following requirements:

a. Satisfy modularity requirements.

b. Make significant use of abstraction, information hiding, and layering in the design and implementation of trusted recovery functions.

6.1.1.2 Trusted Facility Management

Trusted recovery functions shall be assigned exclusively to administrative personnel with security-relevant responsibility, e.g., System Programmer or Security Administrator roles [24].

6.1.2 Life-Cycle Assurance

6.1.2.1 Security Testing

Security testing requirements of class B3 apply to the functions and interfaces of the TCB for user-induced failures (i.e., for state-transition failures as defined in Chapter 2 of this guideline), and to functions and interfaces of administrative roles but only for discontinuities generated by administrative personnel. See discussion in Section 5.2.

6.1.2.2 Design Specification and Verification

DTLSs of the TCB functions and interfaces implementing trusted recovery must be maintained that completely and accurately describe these functions and interfaces in terms of exceptions, error messages, and effects.

a. A formal security model should be used to define the TCB response to state-transition failures (defined in Chapter 2 and discussed in Section 5.2).

b. A formal security model should be used for the derivation of the security policy invariants and constraints used for the design of trusted recovery.

c. Additional invariants and constraints should be used for the design of trusted recovery in the accountability area as needed.

6.1.2.3 Configuration Management

All configuration management requirements of class B3 apply to trusted recovery as stated.

6.1.3 Documentation

6.1.3.1 Trusted Facility Manual

The following items should be included in the trusted recovery section of the Trusted Facility Manual:

a. Procedures for analysis of system dumps, for consistency checking of TCB objects, and for system cold start and emergency restart.

b. A description of the types of tolerated failures and examples of the recommended procedures for responding to such failures.

c. Procedures for running periodic integrity checks on the TCB database and for repairing damaged security labels.

d. Procedures for handling inconsistencies of the system objects (e.g., duplicate allocation of disk blocks to objects, inconsistent object links).

e. Lists of commands, system calls, and function definitions for trusted recovery (whenever these aren't documented in the system's DTLS).

f. Examples of, and warnings about, potential misuse of trusted recovery procedures.

6.1.3.2 Test Documentation

All test documentation requirements of class B3, except those for covert channel testing (viz., Section 5.1), apply to the TCB functions and interfaces implementing trusted recovery as stated. The test plans for trusted recovery should include the following:

a. Test conditions; i.e., a list of discontinuities of operation that can be generated through administrative interfaces and their effects on the system.

b. Test data, consisting of the following:

    (1) Environment setup; e.g., the TCB and user-level data structures and objects needed to generate the planned discontinuity.

    (2) Parameters and commands used by the administrators to generate the discontinuity.

    (3) Expected outcome; e.g., the type of procedures that are started automatically or manually for handling the generated discontinuity and the effect of those procedures on the system state.

c. Coverage analysis; e.g., this includes a list of failures, or classes of failures, whose effect is covered by the generated discontinuities, and a list of spontaneous failures, or classes of failures, whose effect isn't covered by the test.

6.1.3.3 Design documentation

Documentation shall describe the following:

a. Interfaces between the TCB modules implementing trusted recovery functions.

b. Specific TCB protection mechanisms used ensuring trusted-recovery functions are available only to administrative users.

c. DTLS of the TCB modules implementing interfaces of trusted recovery; (Formal Top Level Specifications (FTLS) aren't required for trusted recovery interfaces—viz., relevant discussion in [24] for administrative interfaces).

Design documentation also should include a description of the following:

a. Anticipated classes of failures and discontinuities of operation handled by trusted recovery, automatically or using administrative procedures.

b. Trusted recovery philosophy; viz., Section 5.3.

c. Warnings concerning the "unanticipated" (i.e., rare) failures that can't be handled in a routine manner.

d. State-security invariants and constraints maintained by trusted recovery.

e. DTLS of the TCB primitives implementing trusted-recovery interfaces.

The accuracy of the design documentation should be commensurate with that of other similar documentation for B3 and A1 systems.

6.2 ADDITIONAL REQUIREMENTS OF SECURITY CLASS A1

All requirements of the security class B3 are included here. The only additional requirements are in the following life-cycle assurance areas.

6.2.1 Additional Life-Cycle Assurance Requirements

6.2.1.1 Configuration Management

All additional configuration management requirements of class A1 apply as stated.

6.2.1.2 Trusted Distribution

All trusted distribution requirements of class A1 apply to the TCB functions and interfaces implementing trusted recovery as stated.

GLOSSARY

ACCESS

A specific type of interaction between a subject and an object that results in the flow of information from one to the other.

ADMINISTRATOR

See **Security Administrator.**

APPROVAL/ACCREDITATION

The official authorization that is granted to an ADP system to process sensitive information in its operational environment, based upon comprehensive security evaluation of the system's hardware, firmware, and software security design, configuration, and implementation and of the other system procedural, administrative, physical, TEMPEST, personnel, and communications security controls.

AUDIT

To conduct the independent review and examination of system records and activities.

AUDITOR

An authorized individual, or role, with administrative duties, which include selecting the events to be audited on the system, performing system operations to enable the recording of those events, and analyzing the trail of audit events.

AUDIT MECHANISM

The device, or devices, used to collect, review, and/or examine system activities.

AUDIT TRAIL

A chronological record of system activities that is sufficient to enable the reconstruction, reviewing, and examination of the sequence of environments and activities surrounding or leading to an operation, a procedure, or an event in a transaction from its inception to final results.

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.

CRASH

A system failure that causes the processors' registers to be reset to some standard values.

DATA

Information with a specific physical representation.

DESCRIPTIVE TOP-LEVEL SPECIFICATION (DTLS)

A top-level specification that is written in a natural language (e.g., English), an informal program design notation, or a combination of the two.

DISCRETIONARY ACCESS CONTROL (DAC)

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, or based on the possession of system-protected tickets that contain privileges for objects (e.g., capabilities). 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.

FORMAL SECURITY POLICY MODEL

A mathematically precise statement of a security policy. To be adequately precise, such a model must represent the initial state of a system, the way in which the system progresses from one state to another, and a definition of a "secure" state of the system. To be acceptable as a basis for a TCB, the model must be supported by a formal proof that if the initial state of the system satisfies the definition of a "secure" state and if all assumptions required by the model hold, then all future states of the system will be secure. Some formal modeling techniques include: state transition models, denotational semantics models, and algebraic specification models.

FORMAL TOP-LEVEL SPECIFICATION (FTLS)

A top-level specification that is written in a formal mathematical language to allow theorems showing the correspondence of the system specification to its formal requirements to be hypothesized and formally proven.

IDEMPOTENT ACTIONS

An ordered list of actions (e.g., procedure calls, etc.) is said to be idempotent if repeated incomplete executions of that list of actions followed by a complete execution has the effect of a single complete execution of that list of actions. An idempotent action is a restartable action; i.e., if the action was in progress at the time of a crash, the action can be repeated during crash recovery with no undesirable side effects [14, 21].

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, programs, bits, bytes, words, fields, processors, video displays, keyboards, clocks, printers, and network nodes.

OPERATOR

An administrative role or user assigned to perform routine maintenance operations of the ADP system and to respond to routine user requests.

PASSWORD

A protected/private character string used to authenticate an identity.

PROCESS

A program in execution.

READ

A fundamental operation that results only in the flow of information from an object to a subject.

SECURITY ADMINISTRATOR

An administrative role or user responsible for the security of an Automated Information System and having the authority to enforce the security safeguards on all others who have access to the Automated Information System (with the possible exception of the Auditor.)

SECURITY LEVEL

The combination of a hierarchical classification and a set of non-hierarchical categories that represents the sensitivity of information.

SECURITY MAP

A map defining the correspondence between the binary and ASCII formats of security levels (e.g., between binary format of security levels and sensitivity labels).

SECURITY POLICY

The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information.

SECURITY POLICY MODEL

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

SECURITY TESTING

A process used to determine that the security features of a system are implemented as designed. This includes hands-on functional testing, penetration testing, and verification.

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.

SYSTEM PROGRAMMER

An administrative role or user responsible for the trusted system distribution, configuration, installation, and non-routine maintenance.

TOP-LEVEL SPECIFICATION (TLS)

A non-procedural description of system behavior at the most abstract level; typically, a functional specification that omits all implementation details.

TRAP DOOR

A hidden software or hardware mechanism that can be triggered to permit system protection mechanisms to be circumvented. It is activated in some innocent-appearing manner (e.g., a special "random" key sequence at a terminal). Software developers often introduce trap doors in their code to enable them to reenter the system and perform certain functions. Synonymous with back door.

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. For example, making a "blind copy" of a sensitive file for the creator of the Trojan Horse.

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.

USER

Person or process accessing an Automated Information System either by direct connections (i.e., via terminals), or indirect connections (i.e., prepare input data or receive output that is not reviewed for content or classification by a responsible individual).

VERIFICATION

The process of comparing two levels of system specification for proper correspondence (e.g., security policy model with top-level specification, TLS with source code, or source code with object code). This process may or may not be automated.

WRITE

A fundamental operation that results only in the flow of information from a subject to an object.

BIBLIOGRAPHY

[1] Bach, M. J., *The Design of the UNIX Operating System*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1986.

[2] Baldwin, R. W., *Rule-Based Analysis of Computer Security*, Massachusetts Institute of Technology, Cambridge, Massachusetts, Technical Report MIT/LCS/TR-401, March 1988.

[3] Bell, D. E., and L. J. La Padula, *Secure Computer System: Unified Exposition and Multics Interpretation*, MITRE Corp., Bedford, Massachusetts, Rep. No. MTR-2997, 1976. Available as NTIS AD-A023 588.

[4] Benzel, T. V., and Travilla, D. A., "Trusted Software Verification: A Case Study," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, April 1985, pp. 14-30.

[5] Bina, E. J., and P. A. Emrath, "A Faster fsck for BSD UNIX," in *Proceedings of the USENIX Conference*, San Diego, California, February 1989, pp. 173-185.

[6] Chang, A., M. Mergen, S. Porter, R. Rader, and J. Roberts, "Evolution of Storage Facilities in the AIX System," in *IBM Risc System/6000 Technology*, SA23-2619, IBM Corporation, Austin Communications Department, 11400 Burnet Road, Austin, TX 78758, pp. 138-142.

[7] Chang, A., and M. Mergen, "801 Storage: Architecture and Programming," *ACM Transactions on Computer Systems*, vol. 6, no. 2, February 1988, pp.2-50.

[8] Cristian, F., "Correct and Robust Programs," *IEEE Transactions on Software Engineering*, SE-10/2, March 1984, pp. 163-174.

[9] Cristian, F., "A Rigorous Approach to Fault-Tolerant Programming," *IEEE Transactions on Software Engineering*, SE-11/1, January 1985, pp. 23-31.

[10] Department of Defense, *Security Requirements for Automated Information Systems (AISs)*, DoD Directive 5200.28,21 March 1988.

[11] Gasser, M., *Building A Secure Computer System*, Van Nostrand Reinhold, New York, 1988.

[12] Gligor, V. D., J. C. Huskamp, S. R. Welke, C. J. Linn, W. T. Mayfield, *Traditional Capability-Based Systems: An Analysis of their Ability to Meet the Trusted Computer Security Evaluation Criteria*, Institute for Defense Analyses, Alexandria, VA. IDA Paper P-1935, February 1987; available as NTIS AD-B119332.

[13] Gligor, V. D., "A Note on the Denial-of-Service Problem," *Proceedings of the 1983 IEEE Symposium on Security and Privacy*, Oakland, California, April 1983, pp. 5101-5111.

[14] Gray, J. N., "Notes on Database Operating Systems," in *Operating Systems—An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmuller, eds., Springer-Verlag, New York, 1978, pp. 393-481. Also published as IBM Research Report RJ 2188, February 1978.

[15] Gray, J. N., Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger, "The Recovery Manager of the System R Database Manager," *Computing Surveys*, 13/2, June 1981, pp.223-242.

[16] Hecht, M. S., and Gabbe, J. D., "Shadowed Management of Free Disk Pages with a Linked List," *ACM Transactions on Database Systems*, 8/4, December 1983, pp. 503-514.

[17] National Computer Security Center, *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD, December 1985.

[18] IBM Corp., "Information Management System/Virtual Systems (IMS/VS), Programming Reference Manual," IBM Form No. SH20-9027-2, Section 5.

[19] IBM Corp., *Secure Xenix, Version 1.1—System Administrators Guide*, June 1987.

[20] Israel, J., J. Mitchell, and H. Sturgis, "Separating Data from Function in a Distributed File System," *Proceedings of the Second International Symposium on Operating Systems,* IRIA, Rocquencourt, France, October 1978.

[21] Lampson, B. W., "Atomic Transactions," in *Distributed Systems—an Advanced Course*, B. W. Lampson, M. Paul, and H. J. Siegert, eds., Springer-Verlag, New York, 1981, pp. 246-265.

[22] Lampson, B. W., Robert F. Sproull, "An Open Operating System for a Single-User Machine," in *Proceedings of the Seventh Symposium on Operating Systems Principles*, Pacific Grove, California, December 1979, pp. 98-105.

[23] Mitchell, J. G, and J. Dion, "A Comparison of Two Network-Based File Servers," *Communications of the ACM*, 25/4, April 1982, pp. 233-245.

[24] National Computer Security Center, *A Guide to Understanding Trusted Facility Management*, NCSC-TG-015, version 1, 18 October 1989.

[25] Paxton, W. H., "A Client-Based Transaction System to Maintain Data Integrity," *Proceedings of the Seventh Symposium on Operating Systems Principles*, Pacific Grove, California, December 1979, pp. 18-23.

[26] Saltzer, J. H., "Protection and Control of Information Sharing in Multics," *Communications of the ACM*, vol. 17, no. 8, July 1974, pp. 388-402.

[27] Swinehart, Daniel, Gene McDaniel Boggs, "WFS: a Simple Shared File System for a Distributed Environment," *Proceedings of the Seventh Symposium on Operating Systems Principles*, Pacific Grove, California, December 1979, pp. 9-17.

[28] Walker, S. T., "The Advent of Trusted Computer Operating Systems," *National Computer Conference Proceedings*, May, 1980, pp. 655-665.

[29] Walter, K. J., W. F. Ogden, W. C. Pounds, F. T. Bradshaw, S. R. Ames, K. J. Biba, J. M. Gilligan, D. D. Schaefer, S. l. Schaen, D. G. Shumway, *Modeling the Security Interface*, Technical Report, Case Western Reserve, University, Cleveland, Ohio, August 1974.