Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

ARCHITECTURAL ISSUES IN FAULT-TOLERANT, SECURE COMPUTING SYSTEMS

Mark Kenneth Joseph

September 1988
CSD-880047

TECHNICAL BIOGRAPHY

MARK K. JOSEPH received the B.S. degree in computer science from The State University of New York at Stony Brook, the M.S. and Ph.D. degrees in computer science from UCLA in 1981, 1983, and 1988, respectively.

From 1983 to 1986 he was employed at the System Development Corp., Santa Monica, CA., on the "BLACKER" network security project. From 1986 to 1988 he was a Member of the Technical Staff at The Aerospace Corp., El Segundo, CA., participating in the design and evaluation of fault-tolerant space and ground computing systems. His research interests include fault-tolerant computing, computer security, and computer architecture.

The work contained in this volume presents the opinion of its author, and not necessarily of the organizations he has been, or is currently affiliated with.

UNIVERSITY OF CALIFORNIA

Los Angeles

ARCHITECTURAL ISSUES IN FAULT-TOLERANT, SECURE COMPUTING SYSTEMS

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Mark Kenneth Joseph

1988

© Copyright by

Mark Kenneth Joseph

1988

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First, I would like to thank my advisor Algirdas Avizienis. His encouragement and support of my research ideas made this work possible. His lectures in fault tolerance, his guidance, and friendliness throughout my last years of graduate school are greatly appreciated.

Second, I would like to thank the other members of my committee for their time and suggestions: Kirby Baker, Wesley Chu, David Rennels, and William Yeh. Special thanks to Rein Turn, from Cal State Northridge, for being a guest committee member to fill out the committee's background in Computer Security.

Third, special thanks and warm regards to George C. Gilley of The Aerospace Corporation for the opportunity to work in fault tolerance out in industry, for his genuine concern for my progress at UCLA, for sharing his extensive knowledge in fault tolerance, and for his friendship.

Also, I want to thank the remaining members of the fault tolerance expert team who consulted at Aerospace: Algirdas Avizienis, Jacob Abraham, Bill Carter, and David Rennels. Their willingness to freely share their expertise and for allowing me to comfortably fit into the group has greatly helped this work. This work experience has provided me with a solid understanding of many concepts and issues in fault tolerance, and is greatly appreciated.

Forth, I would like to thank a few good friends. Thanks to a fellow Ph.D. student, Michael R. Lyu, for his extensive help in the malicious logic injection experiments, and for providing information about the specifics of the N-Version Programming support system (i.e., DEDIX). Thanks to Jon King, of The Aerospace Corporation, for reading and questioning a great deal of this work.

His questions and apparent confusion about my ideas motivated many sections in this dissertation. Also, I want to thank my wife Patricia Liu for insisting that I return to UCLA for the Ph.D. degree, and for her constant love and affection.

Last, I want to thank Jim Murrell, my former department manager, and Jim Hamilton, my former section manager from The Aeropace Corporation, for being sympathetic of the demands that graduate work made on my job at Aerospace.

VITA

November 3, 1959	Born, Manhattan, N.Y.
May 1981	B.S. in Computer Science, The State University of New York at Stony Brook
October 1981–March 1983	Teaching Assistant, UCLA Computer Science Department
June 1983	M.S. in Computer Science, University of California, Los Angeles
1983–1986	Senior Software Engineer, System Development Corporation, Santa Monica, California
1986–1988	Member of the Technical Staff, The Aerospace Corporation, El Segundo, California

PUBLICATIONS

Mark K. Joseph, "PROGRAMMING WITH small BLOCKS," ACM Software Engineering Notes, Vol. 9, No. 5, Oct. 1984, pp.28–42.

Mark K. Joseph, "Towards the Elimination of the Effects of Malicious Logic: Fault Tolerance Approaches," 10th National Computer Security Conference, Sept. 1987, pp.238–244.

Mark K. Joseph, and Algirdas Avizienis, "Software Fault Tolerance and Computer Security: A Shared Problem," National Security Industrial Association's Conference on Software Quality and Reliability, March 1988.

Mark K. Joseph, and Algirdas Avizienis, "A Fault Tolerance Approach to Computer Viruses," IEEE Symp. on Security and Privacy, April 1988, pp.52–58.

ABSTRACT OF THE DISSERTATION

Architectural Issues in Fault-Tolerant, Secure Computing Systems

by

Mark Kenneth Joseph

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1988

Professor Algirdas Avizienis, Chair

"It is perhaps surprising that more attention has not been paid to fault tolerance techniques in order to achieve security in computer-based systems" [Dobs86].

To this date, multilevel secure computer systems have not adequately considered fault tolerance as a design goal. In fact, there is a classical bit of wisdom among security analysts, which says that, if the computer system stops, then it is secure since no leakage of sensitive data is possible. However, more than just the concern for leakage of data is becoming important in secure systems, and that is that they provide service in the presence of malicious attacks.

This dissertation explores several facets of the applicability of fault tolerance techniques to secure computer design, these being: (1) how fault tolerance techniques can be used on unsolved problems in computer security (e.g., computer viruses, and denial-of-service), (2) how fault tolerance techniques can be used to support classical computer security mechanisms in the presence of accidental and deliberate faults, and (3) the problems involved in designing a fault-tolerant, secure computer system (e.g., how computer security can degrade along with both the computational and fault tolerance capabilities of a computer system).

The approach taken in this research is almost as important as its results. It is different from current computer security research in that a design paradigm for fault-tolerant computer design is used [Aviz87a]. This led to an extensive fault and error classification of many typical security threats. Throughout this work a fault tolerance perspective is taken (i.e., faults, such as design flaws, are assumed to always exist in a computer system, and that run-time mechanisms are necessary to tolerate them).

However, we have <u>not</u> ignored basic computer security technology. For some problems we have investigated how to support and extend basic security mechanisms (e.g., trusted computing base), instead of trying to achieve the same result with purely fault tolerance techniques.

CHAPTER 1

INTRODUCTION

1.1 THE NEED FOR FAULT-TOLERANT, SECURE COMPUTING

The need for fault-tolerant, secure computing systems is becoming quite evident (e.g., the SDI application). This has sparked the exploration of the issues concerning the design of computing systems that possess both attributes. Fault tolerance and security concerns are not disjoint. For example, security considerations may prevent the use of rollback to recover from an error [Turn86].

Attacks on a computer system can take one of three forms: from completely outside the computing system, from a legitimate, authorized user trying to extend his or her allowed access rights, and from within the computer system itself due to design flaws purposely planted by its designers. It should no longer be acceptable to consider only the classical security concerns of preventing unauthorized disclosure of sensitive information, and unauthorized modification of information and programs. The elimination of the effects of malicious logic must be addressed, and this reveals many similarities to problems in fault tolerance. Two of these effects that this discussion is mainly concerned with are, denial-of-service and compromising integrity.

It is envisioned that in the near future most military computer systems will require both fault tolerance and security properties. Other critical systems may soon follow (e.g., financial, point-of-sale, and airplane reservations) where failures, deliberate or otherwise, are potentially unacceptable due to loss of life, finances, and/or privacy. For example, as of 1987, an average of a trillion dollars in payments, on a typical day, are exchanged by banks over electronic telecommunications networks [FRBS87]. This represents a potential financial disaster if fault-tolerant, secure, and high integrity communications and processing are not guaranteed.

1.2 SIMILARITIES BETWEEN FAULT TOLERANCE AND COMPUTER SECURITY

Several authors have observed many similarities and interrelationships between fault tolerance and computer security [Dobs86] [Turn86]. In fact, in [Dobs86], denial-of-service is viewed as the classical unreliability problem. For example, the concept of fault avoidance (i.e., to prevent, through high quality of hardware and software, the occurrence of faults) can be found in both fault tolerance and computer security.

In computer security the use of formal verification technology [Cheh83] is a form of fault avoidance. Formal verification is used to prevent, and to provide assurances that malicious logic does not appear in secure computing systems. This parallel is complete, since in both disciplines fault avoidance is inadequate by itself.

Additionally, the concept of system partitioning is used in both disciplines. In computer security trusted (e.g., trusted computing base (TCB) [DoD85a]) and untrusted software are separated in order to prevent the untrusted software from causing improper service. This is similar to partitioning in the design of fault-tolerant computing systems. Here, it is also used to prevent errors from propagating and resulting in improper service. However, the main distinction between fault tolerance and computer security is that security is mainly concerned with faults in the fault class of "by intent." Computer security is concerned with deliberate faults while fault tolerance mainly addresses accidental ones [Turn86].

Another important similarity is that malicious logic can be viewed as deliberate faults (see Chapter 2 and 3). Taking this point of view raises a question yet to be asked: "Can we apply techniques from fault tolerance to solve some of the unsolved problems in computer security?" [Jose87] has started to approach this question and its treatment is included in this dissertation.

From these obvious similarities the question which arises (and is discussed in this dissertation) is: "Can one set of concepts and techniques be applied to fault-tolerant, secure computing systems?" [Dobs86].

1.3 SECURITY AS PART OF THE DEPENDABILITY CONCEPT

In [Aviz86] the concept of dependable computing is refined. Dependability is defined "as the property of a computer system that allows reliance to be justifiably placed on the service it delivers." This is a qualitative property consisting of the following components: reliability, availability, readiness, maintainability, testability, and safety.

The achievement of dependable computing is measured via the performance of a system as perceived by its users. It is one of the main theses of this dissertation that security is also a component of dependability. This is also the point of view presented in [Dobs86]. This is due to the observation that lack of security in specific environments can result in improper service. Thus, unauthorized disclosure of sensitive information, unauthorized modification of information or programs, denial-of-service, and compromising integrity can be viewed as errors resulting from some deliberately placed fault (in hardware, firmware and/or software). The failure of a system from deliberate faults is just as dangerous as any others.

1.4 APPLICATION OF FAULT TOLERANCE CONCEPTS TO COMPUTER SECURITY

[Jose87] was a preliminary investigation into applying concepts from fault tolerance to two largely unsolved problems in computer security. These problems are the denial-of-service and compromising integrity threats. Part of this study revealed that malicious logic can be viewed as deliberate design faults and that techniques such as N-Version Programming (NVP) [Aviz85a] can be useful in masking out their effects. Additionally, it was shown that concepts such as program flow monitors [Osde79], and software safety [Leve86] are also applicable.

1.5 OBJECTIVES OF THIS RESEARCH

It is the intent of this research to: (a) determine the issues involved in the design of fault-tolerant, secure computer systems, such as common solutions and incompatibilities, and (b) to explore the application of concepts from fault tolerance to eliminate the effects of malicious logic on computing systems. Experiments with an N-version system were performed to determine the effectiveness of NVP against malicious logic (see Chapter 5).

1.6 OFF-LINE VERSUS ON-LINE TECHNIQUES

Off-line techniques are directed at preventing the insertion of malicious logic (i.e., deliberate faults) for a system's entire life-cycle. Examples of these are formal verification, fault-tree analysis, code reviews, and configuration control. On-line techniques include additional software, hardware, and partitioning methods aimed to counter the effects of malicious logic that has successfully made its way into a deployed computer system.

1.7 FAULT AVOIDANCE IS NOT ENOUGH

Off-line techniques (which are just instances of fault avoidance methods) are severely limited in their effectiveness. Malicious logic can be effective from many locations in a system's software and/or hardware. This large search space provides many opportunities and makes it difficult to prevent or detect malicious logic. Current formal verification techniques and tools can effectively examine only small pieces of software and hardware (e.g., a security kernel in an A1 certified computer system [DoD85a]).

One of the main differences between fault tolerance and computer security is that the later has not fully recognized the limitations of off-line (fault avoidance) techniques as a serious problem. It is a main thesis of this dissertation that on-line fault tolerance techniques are essential. As is also expressed in [Dobs86], run time checks must exist in a system in order to protect the security of a system from some dormant fault. This is not to say that off-line techniques are not needed, just that they can be less effective.

1.8 ISSUES IN BUILDING A FAULT-TOLERANT, SECURE COMPUTER

Pioneering work on this subject appears in [Turn86]. The issues explored were how to make the security mechanisms employed provide proper service in the presence of faults. Additionally, the impact on security when complex fault tolerance mechanisms are added to a computing system is addressed. The significant list of questions posed in [Turn86] are:

- "Are the techniques for achieving fault tolerance and data security fully compatible? If not, what are the problems, and how can they be resolved? What tradeoffs are available?"
- "How does the architectural design for fault tolerance impact the design for security, and vice versa? How can the designs be made compatible?"
- "Can data security be gracefully degrading? Can gracefully degrading systems be data secure? Is there a difference in achieving each?"

1.8.1 CAN ONE SET OF TECHNIQUES SOLVE BOTH PROBLEMS?

It would be cost-effective for a computing system required to be both fault-tolerant and secure to use one set of mechanisms to achieve both. To achieve this, both the similarities and incompatibilities between fault tolerance and computer security must be understood.

[Turn86] explores the question of whether security can be degradable. That is, as certain security mechanisms become unavailable due to failures can the remaining mechanisms still provide some level of security (e.g., B2, C2 [DoD85a]). This is an interesting concept and one that would undoubtedly require proper functional partitioning, and lessons learned from fault tolerance.

It has been alluded to in [Dobs86] [Turn86], and by some preliminary work done in [Jose87], that fault tolerance techniques can be used to tolerate the effects of deliberate faults. Deliberate faults have a parallel problem in computer security that being malicious logic which includes Trojan horses [DoD85a], denial-of-service threat, compromising integrity threat, and others. Also, the addition of fault tolerance mechanisms to a secure system adds to the deliberate faults possible. One such example, is the deliberate invocation of false fault alarms, thus causing some degree of denial-of-service [Turn86].

Certainly, software cannot be guaranteed to be error free, therefore, run time checks should be added to provide assurances of correct operation. Thus, instead of relying on only fault avoidance, computer security (e.g., trusted computing base) can use design diversity to handle accidental design faults [Dobs86].

1.8.2 INCOMPATIBILITIES

In order to actually use one set of techniques for both problems the incompatibilities of both must be determined. Again, [Turn86] has the most recent list. First, many security violations will not be able to be recovered by backward recovery techniques. Once sensitive data has been compromised, bring the system back to a previous state is irrelevant. Another example of where backward error recovery is ineffective, is in firing a missile. Only forward recovery, such as self-destruct is effective here.

Adding fault tolerance mechanisms to a secure system can significantly increase its complexity. Complexity makes it much more difficult to provide assurance of the security of a system. For instance, the fault tolerance mechanisms may provide new covert channels [DoD85a] for data leakage, and new locations for Trojan horses [Turn86] (see Chapter 4).

Often fault tolerance requires maintaining multiple copies of data and/or performing multiple computations with voting. In security the more copies of sensitive data that are around the more vulnerable is that data to disclosure, thus violating the computing system's security policy [Hsia79].

Lastly, for secure systems which handle multiple levels of secure data (multilevel secure) only highly trusted mechanisms are allowed to access all levels of sensitive data. This restriction prevents sensitive data from one sensitivity level from being copied into data of another, likely violating the security of the system. Most fault tolerance mechanisms will have access to data at several security levels, thus requiring it to be trusted. The assurances needed for this type of trust are expensive, difficult to prove, and very time consuming to perform.

1.9 AN EXERCISE IN FAULT TOLERANCE OR COMPUTER SECURITY?

It is likely that the fault tolerance community of researchers will view this research as an exercise in computer security. This should be due to the fact that fault tolerance does not typically deal with deliberate faults, which are addressed in this work. On the other side of the coin, it is also likely that the computer security community of researchers will view this work as an exercise in fault tolerance. This should be due to the fact that this study does not explicitly concentrate on security policies nor security models.

However, this work represents research in both specialties. This can be justified because the discussion herein mainly takes a combined view of the problem (i.e., it does not discuss fault tolerance or computer security separately, but interrelates the two). Also, the references that were relied upon for this work represent some of the most fundamental work in both fault tolerance and computer security.

Specific reasons why this research is an exercise in fault tolerance are: (a) a fault tolerance perspective is used (i.e., fault avoidance is not enough), (b) a fault, error, and failure analysis is performed and is a basis for addressing security threats (see Section 3.2), and (c) standard fault tolerance techniques (e.g., program flow monitors for error detection) are applied to counter security threats, and to propose design choices for fault-tolerant, secure computers.

Specific reasons why this research is also an exercise in computer security are: (a) a wide range of typical security threats (e.g., covert channels, computer viruses) are addressed, (b) some standard computer security techniques (e.g., a trusted computer base (TCB) [DoD85a]) are used along with fault tolerance mechanisms, and (c) some fault avoidance techniques typically used for computer security (e.g., formal verification) are advocated to be used according to there limitations (e.g., formal verification is only effective on small pieces of code) and are complementary with fault tolerance mechanisms.

1.10 A NOTE TO THE PROSPECTIVE READER

The research contained in this dissertation involves advanced concepts in both the computer science subfields of fault-tolerant computing and computer security. Throughout this discussion concepts from each subfield are freely intermixed. As such, this can make it difficult for the reader, without the proper technical background in both subfields, to follow the discussion.

It is envisioned that the readers of this work will have a technical background in fault tolerance or computer security, but not both. There is no desire, nor anything gained, by making this dissertation tutorial in nature, since substantial background material exists in the open literature. However, some help for the prospective reader is essential. In this vein, it is strongly recommended, that the reader review, read, and use as a constant reference, the following references on fault tolerance [Renn84] [Aviz86], and on computer security [Denn82] [DoD85a] [Gass88]. Additionally, glossaries of important terms and concepts are included on fault tolerance and computer security.

CHAPTER 2

INTERACTIONS BETWEEN FAULT TOLERANCE AND MALICIOUS LOGIC

The techniques presented in this chapter are only for highly critical systems. They address the threat of a malicious, but presumed to be trusted, engineer inserting malicious logic into the computing system he or she is developing or maintaining. We do not agree with the basic definitions of closed and open security environments presented in [DoD85c, Appendix C]:

> A closed security environment includes those systems in which both of the following conditions hold true:
>
> a. Applications developers (including maintainers) have sufficient clearances and authorizations to provide an acceptable presumption that they have not introduced malicious logic.
> b. Configuration control provides sufficient assurances that applications are protected against the introduction of malicious logic prior to and during the operation of system applications.

An open security environment is basically the opposite of this.

We believe that <u>no</u> clearance is enough assurance to prevent the insertion of malicious logic. Additionally, some implementations of configuration control can be bypassed by the trusted engineer (e.g., making code changes to previously frozen versions). The justification for this viewpoint is that even people with the highest clearances "sell out." Perhaps the most famous of these cases is the Walker espionage case [Bamf86]. These cases have been so numerous that, for example, 1985 was labeled "The Year of the Spy" (i.e., many Americans in sensitive positions sold out to foreign powers). The malicious logic threat (though not called that) has already made its way into the popular press

[Boor87], concerning its insertion into military software, especially the SDI. We prefer techniques that severely limit the trust of any system or application, hardware and software.

2.1 A MULTI-PRONGED DEFENSE

Malicious logic can disrupt a computer system's normal operation from many locations in its software (i.e., application and operating system code) and/or hardware. This large search space provides many opportunities and makes it difficult to prevent or detect malicious logic insertion by malicious engineers. A multi-pronged defense, composed of off-line and on-line techniques, is proposed to reduce the risk posed by malicious logic. Off-line techniques (e.g., verification) are directed at preventing the insertion of malicious logic for the entire life-cycle of software and hardware. On-line techniques (e.g., execution monitoring) attempt to counter the effects of malicious logic that has successfully made its way into a deployed computer system.

All the on-line techniques presented in this chapter are completely application dependent, whereas the off-line techniques can be more general. The reason for this lies in the fact that the on-line techniques are used in single threat-countermeasure pairs, whereas each off-line technique can cover many threats.

Tradeoffs of performance versus the degree of risk are essential and should be carried out from the onset of a project. Additionally, to determine the effectiveness of a chosen collection of ad hoc techniques, an error seeding approach directed by penetration teams could be used. Each of these topics is examined in this dissertation.

Other recent work addressing protection techniques against malicious logic appears in [Denn86] [Pozz86] [Sche86] [Lai88]. These techniques could be used in conjunction with those presented here.

2.2 PROPERTIES OF MALICIOUS LOGIC

It is useful to extend the notion of malicious logic, as held by the security community, by giving it a fault tolerance definition. As such, malicious logic can be defined as deliberate design, specification, or algorithm faults, and computer virus errors[1] (see Section 3.4.2) implanted in a computer system with the intent to cause loss or harm to the user (a human or another system). This new definition provides a perspective that motivates the use of fault tolerance approaches to counter malicious logic in all its forms.

Classical examples of malicious logic are Trojan horses, trap doors, and computer viruses. However, other forms of malicious logic are a more general threat than these examples. Specifically, malicious logic can be similar to trap doors, but can cause harm from many places in the system, not just from the TCB (some security mechanisms will be placed outside the TCB, see Section 2.2.3). Also, it can be similar to Trojan horses, but to be effective it does not have to contain an added hidden function, it just has to force a malfunction of the software or hardware's intended function, and it can be implanted by a trusted engineer. Two possible effects of malicious logic, which are addressed throughout this dissertation, are denial-of-service and compromising integrity.

It should be noted, that in [Turn86] the deliberate nature of faults was

[1]This is a fault tolerance characterization of viruses. The virus error is the modification of a program. Any subsequent malicious actions by the virus code (e.g., deleting files), are errors propagated from the initial error stage.

mentioned and that such faults are believed to be very complex. Thus, design faults, accidental or deliberate, are a shared problem for both fault tolerance and computer security.

Malicious logic is designed to avoid detection by both off-line and on-line techniques. To escape detection by off-line techniques, malicious logic is typically hidden in the complexity of the system's software and/or hardware (i.e., large size, sophisticated algorithms, confusing software and/or hardware layout). For example, in software this can be done by the use of multiple levels of macro calls, and by the deliberate use of improper and tricky coding practices (see Chapter 5 and Appendix B).

To hide from on-line techniques, malicious logic can try to create errors which appear to be results of naturally occurring faults. For example, malicious logic could deliberately cause errors in network messages by flipping bits after a checksum has been calculated, or by simply using an incorrect checksum, to achieve a higher error rate and to impair the performance of the network.

Malicious logic need not cause major system malfunctioning to be effective, since just discrediting confidence in a system can be very disruptive (e.g., the SDI [RADC87]). Under this scenario, malicious logic can be simpler than usual, and thus, likely easier to hide.

2.2.1 DENIAL-OF-SERVICE THREAT

As presented in [Glig83] [Glig85a] a service is a general term used to represent the ability to access data, the ability to access programs (e.g., to edit), the ability to execute programs, and the ability to use hardware resources (e.g., a network path, a printer, a CPU). Of course, this includes that access to any object has first been authorized.

The correct functioning of a process or hardware component (e.g., for missile targeting, bank account transactions) is a service. If the process malfunctions due to some deliberate malicious act, then the proper functioning of that process is denied to the using subject. [Glig83] terms this a "misbehaved service," but limits the deliberate act to a malicious user exploiting flaws in access mechanisms or policies in order to modify the service (e.g., a computer virus). By analogy, the correct value of a data item is also a service (e.g., forcing stale, that is, old data into a file [Glig83]).

Denial-of-service is a failure, with the property, that at the service boundary the intended user is prevented, for some amount of time [Glig83], from using the computing system as was specified. This can take many varied forms some of which include: the hoarding of system resources (e.g., CPU, memory) so they are not available when needed, incorrect operation (e.g., never allowing a user to invoke an editor), locking out all accesses to part of a database, omission of action, timeliness of action (e.g., doing something too late), and forcing the machine to completely stop functioning. In short, many different types of errors can result in this same type of failure.

Definitions, examples, and derivation of fundamental principles of denial-of-service in operating systems and computer networks appear in [Glig83] [Voyd83] [Cerf85] [Glig85a] [Voyd85b] [DoD87] [Yu87] [Yu88]. It is important to recognize that several of the ideas presented in [Glig85a] support the application of fault tolerance concepts to the denial-of-service threat. These ideas include the detection of, and recovery from, denial-of-service (see Section 3.4.1).

2.2.2 COMPROMISING INTEGRITY THREAT

In [Port85] several definitions for integrity in a security context are presented, the pertinent ones are:

1. How correct (we believe) the information in an object is.
2. How confident we are that the information in that object is actually from the alleged source, and that it has not been altered from its original form.
3. How correct (we believe) the functioning of a process is.
4. How confident we are that the functioning of a process [or any software or hardware] is as it was designed to be.
5. How concerned we are that the information in an object not be altered.

Each item above is referred to by: "integrity-#." Integrity-2 and integrity-5 are classical concerns, since these are what the current state-of-the-art can ensure [Biba77] [Voyd83] [GC84].

[Port85] later continues with the very interesting statement that "..., we first eliminate integrity-3 and 4, until we have a way to deal with design issues." Integrity-1 is also bypassed for approximately the same reason. However, through the use of fault tolerance techniques we plan to demonstrate that integrity-1,3, and 4 can be supported. Integrity-3 and 4 from a fault tolerance perspective involves ensuring proper service of a function (software or hardware) with respect to a defined fault class. Integrity-1 can be partially provided by preventing a failing function from generating incorrect data (e.g., for missile targeting).

So we designate the items integrity-1, 3 and 4 as ensuring "integrity of function and data." Two examples of violations to be avoided are: inaccurate (old) data deliberately placed into a database, and incorrect actions (e.g., fire a missile at an ally). In Section 3.2, we go into detail to define a set of examples which delimit these concerns.

RELATION TO OTHER INTEGRITY CONCERNS

Integrity of function and data is not the classical integrity concerns presented in [Biba77] [Voyd83]. Those address the unauthorized modification of programs and data, and the authentication of a data item's source (i.e., integrity-2 and 5 above). (See definitions of integrity level, integrity policy, integrity simple condition, and integrity *-property [pronounced star property] in the glossary for computer security terms and concepts.)

The integrity concerns presented in [Clar87] [WIPC87] are relatively new to security technology and so a brief description is needed. Basically, this type of integrity can be viewed as data (can be extended to all objects) being controlled by a type manager. The data is encapsulated by the type manager in the following ways: (a) operations on a data item can only be performed by programs defined by the type manager, (b) an accessing user (can be extended to all subjects) has limited access to those operations defined by the type manager (i.e., in [Clar87] users are allowed to execute only a few of the defined operations in order to prevent fraudulent manipulation of the data), (c) an operation can define additional constraints on its use (e.g., what time of the day it is allowed), (d) the integrity of the encapsulated data item is periodically validated by verification programs, which are also defined by the type manager (i.e., they detect an invalid state), and (e) the type manager must be protected from unauthorized modification (i.e., its programs and control data).

Essentially, this type of integrity policy restricts what a user (subject) can do to data (object) that it is authorized to modify. This is not possible in the Biba integrity policies. (Of course, it also prevents unauthorized access.)

It is also the aim of integrity of function and data to prevent any software and/or hardware that is authorized (or unauthorized) to modify an object in a malicious way. It goes beyond the type manager approach, because it addresses the design correctness of the operations on data provided by the type manager by the use of run-time mechanisms.

2.2.3 TRUST IN NETWORK SECURITY

Trust that components of a computer system will not violate its security policy is relied upon to build current day secure systems. Typically, such trust is backed by assurances of correctly built components (e.g., development in a closed security environment, use of formal verification). However, the current viewpoint on the degree of trust needed to protect against loss of integrity and denial-of-service in networks is <u>excessive</u> and is not adequately ensured.

The following excerpt from [DoD87, pp.149-150], for class B3 and A1 secure computer networks, states this over reliance on trust.

> It should be clear that some integrity and denial of service features can reside outside the NTCB. [Network Trusted Computing Base] Otherwise all software in a network would be in the NTCB. Every piece of software that has an opportunity to write to some data or protocol field is "trusted" to preserve integrity or not cause denial of service to some extent. For example, it is necessary to "trust" TELNET to correctly translate user data, and to eventually transmit packets. FTP also has to be "trusted" to not inappropriately modify files, and to attempt to complete the file transfer. These protocols can be designed, however to exist outside the NTCB (from a protection perspective). It is beneficial to do this type of security engineering so that the amount of code that must be trusted to not disclose data is minimized. Putting everything inside the NTCB contradicts the requirement to perform "significant system engineering ... directed toward ... excluding from the TCB modules that are not protection critical," .... If everything has to be in the TCB to ensure data integrity and protection against denial of service, there will be considerably less assurance that disclosure protection is maximized.

This excerpt indicates that the notion of a NTCB and the use of trust are inadequate solutions to the possible effects of malicious logic in a network context. This occurs for the reason stated in Section 1.7, that is, the relevant logic that could cause integrity or denial problems is just too big for current fault avoidance techniques to be effective. Thus, to protect against such threats fault tolerance techniques, as presented here, are required.

2.3 APPLYING FAULT TOLERANCE (ON-LINE) TECHNIQUES

The application of fault tolerance techniques to the problem of malicious logic is derived from the observation that its effects can be classified under the fault class of "by intent." This class of faults includes both accidental and deliberate faults.

On-line techniques include additional software, hardware, and partitioning methods aimed at preventing the effects of existing malicious logic in a deployed computer system. It would be useful if, during execution, these techniques could also explicitly determine the location of such logic. Once located, it could then be targeted for removal as soon as possible.

2.3.1 CAN THE EFFECTS OF MALICIOUS LOGIC BE MASKED OUT?

N-Version Programming (NVP) is an approach that aims to provide reliable software by means of design fault tolerance [Aviz85a]. $N > 2$ versions of one program are independently designed and implemented from a common specification (or even from two or more specifications). All N versions are executed concurrently, typically on an N-processor computer system. During execution, the versions periodically form a consensus on intermediate results and on the final result. As long as a majority of versions produce correct results, design faults in one or more versions will be detected and masked out. The strength of this approach is that reliable computing does not depend on the total absence of design faults.

A natural extension of this approach is to employ NVP to maintain the integrity of function and data by masking out the incorrect outputs of deliberate design faults. The probability of identically behaving versions of malicious logic appearing in a majority of the N versions of programs is diminished due to the independent design, implementation, and maintenance of multiple versions (i.e., planting more than one mole [malicious engineer] in many environments, especially a classified one, is difficult).

In Section 2.3.3, and 3.4 an extensive look into the characteristics of and solutions to the denial-of-service threat are presented. Here, it is appropriate to introduce the idea of using NVP against denial-of-service. Instances of denial-of-service threats which involve the hoarding of system resources (e.g., CPU time, disk space) may be prevented by NVP. The specification(s) of the N versions must clearly state a set of restrictions that all versions must adhere to. For example, it can be specified that a version can have only a limited number of open files and/or child (forked as in UNIX)[2] processes. (Each child consumes CPU time, main memory, and disk space.) Now, the voting mechanism used in NVP can be applied to a version's actions, such as system calls made, rather than to generated data values only (see Section 5.1.1). Thus, if less than a majority of versions try to obtain excess resources, the remaining versions will prevent such hoarding by masking out the resource requesting system calls.

It is important to note, that voting on a version's actions should not exces-

[2]UNIX is a trademark of AT&T Bell Laboratories.

sively decrease the possible diversity between versions. This is an important consideration, since an N-version system may be used in an application that requires both fault tolerance and computer security. This issue is discussed in Chapter 5.

A new instance of the denial-of-service threat may be possible for 2VP systems. Malicious logic need only be placed in one version, and would be designed to deliberately cause the two versions to disagree. Typically, a majority of versions is needed in order to produce a result. Thus, continued disagreement could cause some degree of denial-of-service.

At least two solutions exist for this new problem. The above example emphasizes an important feature of most NVP systems, that of masking. Only when N is greater than or equal to three can incorrect actions be masked out. Thus, one solution is to prohibit the use of 2VP systems.

Another solution is to use a hybrid form of NVP and Recovery Blocks [Aviz84] [Aviz85a] to prevent the malicious version from voting and forcing a disagreement. This is done by adding trusted self-checking code (i.e., the acceptance tests used in Recovery Blocks [Rand75]) to both versions. Acceptance tests are additional program statements that are used to test whether a section of code performs as it was specified. Each time the malicious version failed an internal acceptance test its outputs would be ignored, thus preventing the denial-of-service. Such a hybrid form has already been shown to be effective for handling non-deliberate design faults in NVP systems [Aviz84].

It is noteworthy that NVP also addresses completeness which is part of integrity concerns, and timeliness of action. Several versions ensure (through consensus decision) that all specified actions are performed. A timeout mechanism at all decision points prevents prolonged periods without action (e.g., slowing down a computer system can lead to denial–of–service [Glig83]).

Additionally, the acceptance tests in the hybrid form of NVP and Recovery Blocks could attempt to distinguish deliberate and non–deliberate design faults. Thus, detection of deliberate design faults could be used to trigger an alarm notifying the appropriate authorities. It appears that more than just masking out design faults is needed if locating deliberate design faults is also desired. It should be made clear that in general all design faults are important. However, the discussion in this chapter concentrates only on deliberate ones.

NVP is application dependent in two ways. First, determining how much, and which parts, of a software system will be built using NVP may be different for each application. Second, if used against denial–of–service threats, then restrictions placed in the specification(s) will likely be different for many applications.

BACKGROUND DEVIOUS ACTIONS

Can a Trojan horse, inadvertently used by an NVP system, perform devious actions in the background while producing valid results to be voted on? The following points can be made.

First, the whole idea of NVP is that many of a version's actions (e.g., calls made as well as data generated) are voted on (via some decision function). Thus, these background devious actions will either be masked out entirely or severely limited. An obvious tradeoff between degree of risk and performance exists here.

Second, input to each version of an NVP system needs to be obtained from different sources. If each version obtains the same bad data, then the masking capability of NVP could be defeated. By analogy, if each version of an NVP system calls one version of a common program that contains a Trojan horse, then masking out its devious actions will not be possible.

Third, a multi-pronged defense is advocated. Thus, a collection of on-line and off-line techniques should be used. If one technique fails to detect and prevent a devious action then it is hoped that others will catch it. This concept is very similar to the idea of hierarchical error recovery in fault tolerance [Wens78] [Renn84].

2.3.2 CAN THE EFFECTS OF MALICIOUS LOGIC BE DETECTED AND RECOVERED FROM?

In this section we examine several techniques which could be used to detect malicious behavior from any software in a computer system. Additionally, for these techniques to be effective they must include provisions for recovery from the undesirable effects of malicious logic.

SOFTWARE SAFETY

Software safety techniques [Leve85] [Leve86] have been applied to safety critical systems. An entire system view is taken in applying these techniques (i.e., both computer and non-computer hardware). System conditions which could lead to unsafe failures, called hazards, are hypothesized. Fault-tree analysis is used to locate where, if at all, in the system's software these series of conditions could occur.

Safety assertions are used to detect hazards and are a form of the acceptance test used in Recovery Blocks. Safety assertions are placed in the software along with recovery routines which are used to restore a system to a safe operating or fail-safe state. The strength of this method is in the total system view taken.

These techniques are also applicable to prevent the effects of malicious logic. A typical example of denial-of-service is an overloaded use of a system's processing resources (e.g., CPU time). Here the unsafe failure state is denial-of-service, while the hazard is the overloaded processing.

Assume (for this example) that fault-tree analysis determines that in the executive's scheduler this hazardous state could be observed. To counter this threat, the safety assertion appearing in Figure 2.1 could be placed in the scheduler.

```
assert underload: if utilization <= max_limit
       on failure do
           assert diagnosis1: [condition]
           assert diagnosis2: [condition]  od
```

Figure 2.1 Safety Assertion to Detect Overloaded System Use

When the "underload" assertion becomes false, special recovery routines will be invoked via the "on failure do" clause. These routines are application dependent and can be grouped in a safety executive as described in [Leve85].

To handle this possibly intentional overload condition the recovery routines would preempt running application tasks.[3] This should continue until the load on the system's computing power decreases to a point where real work can progress.

MONITORS

Let us consider a large banking institution's transaction processing system

[3]Essential security functions, such as audit, should never be preempted in such a manner.

as a target of malicious logic. In the peak of business activity, the bank's computer network of automated teller machines and mainframes is forced into a self-test operational mode. These tests could require such a significant amount of computing power that the bank's computers are unable to process any significant number of incoming transactions.

This situation could result in a large financial loss to the bank in question [WSJ87]. In fact, the bank could be held for ransom, such that its computers would occasionally be rendered inoperative unless a sum of money were paid. To counter this particular threat and, possibly, others like it, a trusted computing base (TCB) can be defined that mediates actions which are meaningful at the application level [DoD85a, p.67]. Access to objects involves not only reads and writes, but how and when application and operating system functions are invoked. Here, programs are the objects, and access to them is equated to their execution.

Now, invocation of the self-test function can be accomplished only after the TCB scrutinizes the request. All such potentially damaging use of basic system functions can be placed behind this defined security perimeter. All requests which are disallowed can then be viewed as auditable events. This technique requires defining a different security perimeter for each application. The potential for misuse of system functions is typically different for each application.

The signature concept used in program flow monitors (PFM) [Mahm88] can be extended in order to prevent incorrect actions of a program on data items. To do this, each of the defined data manipulation functions (e.g., remove network packet header) is given a unique signature; for example, a sequence of bits in a bit vector. Also, each data item is initially given an empty signature.

The result of a sequence of data manipulations is a combination (i.e., logical AND) of all the performed functions' signatures stored in the data item's signature.

For each sequence of acceptable data manipulations, an associated sequence of acceptable signatures exists and is stored in the PFM. That is, one signature exits for the result of each data manipulation. At the application of a data manipulation function, the PFM precomputes what the resultant signature will be if the operation is performed. This precomputed, dynamically generated signature is tested by the PFM to see if it represents a valid signature. If the signature is not acceptable, then the data manipulation is not performed and some response depending on the particular application is necessary (e.g., auditable event, drop packet).

To strengthen this approach, the number of times that the same function is applied to a data item can also be encoded in the signatures [Osde79]. This is done by treating a data manipulators's signature as a number and adding it to the data item's signature. However, a problem with this, is that an invalid sequence of data manipulations can generate a valid resulting signature sum. For example, given signatures with values 2 and 3, three applications of the data manipulation with signature 2 will result in the signature sum of 6, which could be misinterpreted as two applications of signature 3 (i.e., assuming that three applications of an operation with signature 2 is invalid).

To handle this problem two conditions must be met: (a) one signature cannot be a multiple of any other, and (b) invalid applications of a data manipulator cannot generate the same sum generated by a valid sequence of operations. To achieve condition "a," requires all data manipulation signatures to be relatively prime. For condition "b," all common multiples of each pair of signatures are disallowed, or one of the multiples before a common multiple is disallowed (i.e., given signatures 2 and 3 as above, either 6i or 4i is disallowed, where i = 1,2,3,...).

An example of where this can be used is a network protocol function that removes a packet header. Correctly functioning protocol software should remove the header only once. However, malicious logic may try repeatedly to remove the header in order to obtain a packet's data. Assuming that the protocol software is not authorized for access to the packet's data, such access would generate an invalid signature. Thus, program flow monitors can be used to ensure the integrity of function and data. This could also be viewed as just another example of part of a TCB, as mentioned above.

2.3.3 REDUCING THE RISK OF DENIAL-OF-SERVICE IN COMPUTER NETWORKS BY USE OF REDUNDANCY

The scheme presented in this section is specifically designed to prevent some of the intruder-generated denial-of-service [Glig85a] attacks possible in a computer network. An intruder could be an authorized user of a switching node (e.g., IMP in ARPANET) and/or of a gateway, malicious logic such as a Trojan horse program in a switch, or even a party tapping a communication link. An attack could involve the deliberate flooding of a network path with packets, and/or the deliberate dropping, delaying, or modification of packets.

The basic idea is to deliberately transmit duplicate network packets in a datagram network, at the same time, and on different routes in order to prevent an attacker from denying message delivery from any source to any destination host. The standard sliding window protocols used for flow-control in the transport layer (e.g., TCP [Cerf74] [DoD83a]), will automatically accept the first valid message (i.e., in TCP a byte stream) while ignoring all deliberately transmitted duplicates. Obviously, this scheme is only applicable to networks that contain multiple paths between most of the nodes in the network. Typical networks with this property are long-haul ARPANET type, and multiconnected local area networks such as the rings presented in [Ragh85].

It is also immediately obvious that the duplicate packet scheme could cost a great deal in network bandwidth and buffers. Therefore, two specific applications are foreseen for its use. First, some high priority communications have a well defined time limit of significance. For example, a network packet indicating the Japanese attack on Pearl Harbor would have had a small real-time limit, after which a military response would have been too late. Second, it is noted in [Cerf85] that for a network to recover from denial-of-service all forms of network control (e.g., routing update packets, checksums) must be protected from subversion. Thus, control packets can be transmitted using the duplicate packet scheme (and are also given a sequence number). The cost of the duplicate packet scheme is too high for general use in a computer network.

A similar scheme of transmitting duplicate packets on different network routes appears in [Koga82]. This scheme differs from the one presented here in the following ways: (a) in [Koga82], each packet is fragmented into several blocks, and transmitted over a different route, (b) each of these blocks is transmitted redundantly (2 copies) again over separate routes, (c) no consideration is presented on how to implement such a scheme with typical long-haul network protocols (as is done here), (d) the scheme presented in this section, would use encryption techniques rather than packet fragmentation to prevent information disclosure, and (e) [Koga82] is not specifically concerned with maliciously caused denial-of-service in communication networks.

TYPES OF THREATS ADDRESSED

The duplicate packet scheme addresses all cases of denial of message delivery due to dropping or delaying of packets. However, it does not address message stream modification (see active wiretapping definition 2), and thus, must be used with the schemes presented in [Voyd83] [Voyd85a] [Voyd85b].

Physical attacks directed towards causing denial of message delivery on a geographically dispersed network would be relatively easy. These attacks include sabotaging (e.g., destroying) switching nodes, telephone switching stations, microwave stations, and telephone lines that can make up a network. Additionally, it covers tapping of communications links. Active taps could be used to increase link noise in order to corrupt data packets traveling on telephone lines. Local area networks which span an entire building are certainly targets for a mole working inside an organization. Ethernet cables are typically wired unprotected in false ceilings.

Attacks originating from a user or malicious logic in a switching node could delay a packet past its allowed time to live, as in IP [DoD83b], for example. Initially, it seems that basic retransmissions could overcome this deliberate and malicious loss of packets. However, an attacker could achieve denial of message delivery by dropping, delaying, or modifying enough packets in a row so that the combined retransmission times add up to the time needed to achieve the denial. In fact, in the ARPANET, a datagram can repeatedly traverse the same route over several retransmissions, thus, allowing repeated attacks on the same packet by a stationary attacker. To defeat the scheme in this section would require intercepting all the duplicates transmitted on different routes, a feat not likely to be easy.

Deliberately flooding a particular part of a network [Glig85a] could also result in dropping and delaying of packets due to the burden on network resources and to flow-control mechanisms limiting traffic. This attack could be effective in denying one or more network routes from being used, but would be unlikely to prevent all duplicates transmitted from reaching their destination. This again assumes a network with sufficient multiple paths so that at least one duplicate packet could be routed around the congested area. If this is not the case, then denial of message delivery would occur. This is not to say that the duplicate packet scheme is to replace standard flow-control mechanisms [Gerl82]. It is used to supplement them in the presence of deliberate faults.

It seems unlikely that some type of priority scheme can be devised so that packets generated to deliberately congest the network would be dropped. The reason for this, is that, there is no guaranteed distinguishing factor between packets that indicate which were maliciously transmitted. Schemes based on learning that certain hosts contribute excessively to network congestion will likely be too slow to prevent the denial-of-service. Additionally, the attackers could periodically move around the network.

A nice feature of deliberately transmitting duplicate network packets, is that multiple concurrent attacks could be countered at once. As long as one of the packets reaches its destination the attack is defeated. An interesting observation to make is that we can view the duplicate packet scheme from a fault tolerance perspective. Then, we can characterize it as tolerating multiple, coincident, deliberate faults by the use of redundancy of data, the sliding window protocol, and message stream modification protection mechanisms to ignore redundant and corrupted packets.

The duplicate packet scheme will not by itself be able to counter network-wide attacks. An example of this is if all the outgoing links from one source host are destroyed. Another example is when every or most nodes in the network are programmed with malicious logic which is designed to prevent packets from one source to reach a particular destination (or any variant of this [e.g., all packets from one source are to be dropped]).

Traditional routing algorithms will route packets around down sites and even congested areas of a network [Gall77] [Tane81]. What is of importance here, is the time it takes to achieve packet delivery in the presence of deliberate physical and logical faults. If detection of transmission loss and rerouting are fast enough to guarantee packet delivery, in a malicious environment, and before some defined maximum time, then the scheme presented here is redundant.

DELIBERATELY TRANSMITTING DUPLICATE PACKETS

The implementation approach chosen is designed to keep the fact that duplicate packets are being sent transparent above the network layer. The network layer is chosen to handle transmitting duplicate packets in order to ensure that each packet is sent over a different route. The obvious advantage of this approach is that no protocol above the network layer need be modified (e.g., TCP [DoD83a]).

The scheme is as follows. First, the source host transport layer protocol (TCP) hands a message (segment) to the network layer interface. If necessary, the network layer protocol (e.g., IP [DoD83b]) will break the message into several network packets (i.e., transport layer messages can be several packets in size, or may even be of equal size). Second, the network layer protocol is extended to transmit the packet repeatedly the required number of times (i.e., up to the number of different routes to the destination). A distributed, adaptive, multipath, network layer routing algorithm, similar to the one presented in [Gall77], routes each duplicate to the destination host over separate paths. Third, at the destination host the sliding window protocol used by a transport layer connection only accepts the first valid received message, while simply dropping all the duplicates.[4]

In the multipath routing algorithm, each switching node has a routing table that can contain several outgoing link entries for each final network destination. For example, to destination B the three outgoing links L_1, L_2, L_3 could be used. Associated with each L_i entry, is a fraction (F) of traffic that would be transmitted over that link towards the final destination, and where $\sum F = 1$ per destination (e.g., B: (L_1, .10), (L_2, .50), (L_3, .40)). At each switching node, the routing decision is made that picks one of the outgoing links to transmit the packet on. Thus, if several duplicates cross paths at an internal switching node they would again be transmitted along different paths, if possible. This would occur if the multiple paths between a source and destination are not completely disjoint. (If the network layer supports source routing [e.g., IP at the gateway level], then path intersections may be minimized.) Setting F evenly among each outgoing link will evenly distribute the duplicate packets over all available paths.

Sliding window protocols allow several packets (bytes of a message for TCP) to be in transit and unacknowledged at the same time. This is done by defining a subset or window of sequence numbers for packets (or bytes) in

[4]Note, for example, that TCP will treat deliberate duplicate byte streams the same way as accidental duplicates.

transit out of a defined maximum range of sequence numbers. The window moves along the range of sequence numbers as acknowledgements for packets are received. This window defines which packets and how many packets can be sent and received at any instance in time [Cerf74] [Tane81]. After a valid packet is received any duplicates that arrive with an identical sequence number will be outside the receive window, and thus, simply dropped.

Since the multipath routing algorithm is distributed (i.e., each switching node builds its own routing tables based on periodic updates of network state from neighboring nodes [Gall77]), it is resistant to single point of failures and intruder-generated denial-of-service. Obviously, central site routing algorithms are susceptible to denial-of-service (i.e., switch over to a back up routing site may take too long).

Additionally, the routing algorithm could be extended to use a subset of the available multiple paths between a source destination pair. Periodically, a different subset of paths could be switched to in order to prevent an attacker from being sure which paths were currently being used. Switching to alternative routes could be signaled on certain routing information updates.

The duplicate packet scheme presented above can be used by itself, or incorporated with a detection of denial of message service mechanism as presented in [Voyd85a] [Voyd85b]. Once the "request-response" protocol detects the denial-of-service, the duplicate message scheme could be used as part of the recovery action. This will ensure that subsequent message transmissions will not be denied. After some time interval, duplicate message transmission can be automatically turned off, until the next denial-of-service detection.

2.4 APPLYING FAULT AVOIDANCE (OFF-LINE) TECHNIQUES

Off-line techniques are used to remove malicious logic throughout the entire life-cycle (e.g., the development, testing and maintenance stages of a project). Removal of malicious logic follows its explicit detection in the software or hardware of a computer system.

These techniques are applied to all types of software in a computer system (e.g., both operating system and application code). In particular, the on-line security mechanisms chosen to protect a system from attacks are themselves targets for malicious logic insertion. The trust placed in these mechanisms must be validated. This can be done by one or some combination of the following methods: fault-tree analysis as mentioned above, formal verification, testing, simulation, and code reviews.

2.4.1 WEAKNESSES OF CURRENT FORMAL VERIFICATION TECHNOLOGY

In designing secure computer systems, it is important to understand the limitations of all available security technology. This measure of effectiveness can aid in guiding the degree of reliance on each security mechanism or assurance. This section highlights several limitations of current formal verification technology applied to software and hardware. The overall purpose of this section is to dispel any belief that verification (fault avoidance) is a panacea to all computer security problems.

First, the current accepted practice of formally verifying the TCB, does not include verifying system initialization nor built-in-test code. It is simply assumed that formal verification should only prove that from an initial secure state all following state transitions are security-persevering. However, this ignored software is a perfect location for malicious logic insertion. It is important to note, that whereas code correspondence (i.e., matching parts of the specification to the actual code) identifies all code not contained in the specifications, it does not analyze this code in detail.

If malicious logic was disguised to appear as valid software or part of valid software (see Chapter 5), then it may go undetected by the code correspondence process. For example, malicious logic can take the form of a simple extra term added onto a pre-existing and correct equation. This extra term would contain a simple trigger (e.g., x10 is a variable alternating between 0 and 1 in: correct equation + (x10*5.0)), which would control when the equation's result will be incorrect. Another example, could be to force an off-by-one error in a program loop of the initialization code. If a descriptor based machine was used, for example, then this could be designed to write a privileged descriptor over a less privileged one when the system was booting up and configuring access rights. This off-by-one error would also be trigger enabled. Whereas, the above examples could be detected by verification, they will be hard to find by inspection.

Second, to ensure that malicious logic could not be the cause of certain undesired system states (e.g., denial-of-service), large amounts of software and hardware would have to be verified. This is currently beyond the state-of-the-art and will be for some time [Wing86] [Youn87]. Additionally, wanton use of this technique is very likely to lead to extremely expensive developments.

Third, program handling tools such as compilers, editors, and loaders, are currently not themselves verified. As such, they can easily insert malicious logic into the code that they manipulate, thus bypassing formal verification assurances [Thom84] (see Section 3.6).

Fourth, specifications used for formal verification are an abstraction of the actual implementation (both for software and hardware). This provides the possibility that malicious logic in the implementation is missed, either by human error or due to detail being abstracted away.

Fifth, recent experience has shown that the act of writing a formal specification and performing the code correspondence, uncovers many existing flaws or design faults. Far fewer are found by the actual proof process.

Sixth, what happens to the assurances provided by formal verification as the computer system changes throughout its life time? As parts of the system are changed (both software and hardware), parts of the formal proofs must be redone. In a real world environment of deadlines this runs strongly counter to schedules. If not carefully controlled, then the result can be more abstraction as the pressure to do the best possible job in the time allotted is pursued, rather than performing the complete job.

Seventh, problems with current hardware formal verification technology for computer security are presented in Section 3.6.

2.4.2 THE USE OF FORMAL VERIFICATION TECHNOLOGY

Current formal verification techniques and tools can effectively examine only small pieces of software [Youn87] (e.g., a security kernel in an A1 certified computer system [DoD85a]). If the security mechanisms used are too large, then formal verification can be done on selected pieces.

For NVP, formal verification or any validation method should be concentrated on the support software, since this is where malicious logic could have its effects. For example, parts of the DEDIX [Aviz85a] [Aviz85b] (DEsign DIversity eXperiment) system developed at the UCLA Center for Experimental Computer Science should be formally verified (e.g., the voter logic). If each version of the support software was itself from a diverse design, then the importance of verification could be reduced.

For software safety techniques, the safety assertions and recovery routines are candidates for formal verification. Finally, the same extensive methods used for TCBs seem to apply to all types of monitors.

2.4.3 TESTING

Malicious logic could be designed to trigger on particular state conditions (e.g., the date, or number of enemy targets seen) [Myer80]. The trigger could also be disabled until a command was sent enabling it. This enabling command could simply be a sequence of legal but odd system requests (e.g., one hundred health status system requests in a five-minute time interval) (see Section 5.2.3).

Standard testing methods would likely be ineffective in locating such malicious logic, since they would probably miss enabling the triggers. Therefore, new testing approaches aimed at detecting possible enabling command sequences (i.e., channels) and trigger devices should be used. This requires a separate test plan from the normal functional testing.

In addition, the use of independent testing teams from alternate contractors has been shown to increase testing effectiveness. Component testing, at the module level by the independent teams, also seems necessary, since testing at only the device level is of insufficient depth for our purposes.

Alternatively, computer security surveillance technology may be effective in detecting suspicious actions used to enable triggers during normal system operations [Denn86] [Clyd87]. This system monitoring would be useful if it can be done in real-time (i.e., both data collection and analysis) with reasonable performance impact, high detection coverage, and small detection latency.

2.4.4 CONFIGURATION CONTROL

Very strict configuration control software and procedures are essential. This will help to ensure that malicious logic is not added after all tests are made to ensure its absence. To guarantee that proper procedures are followed, surprise inspections could be used in order to monitor the developer.

2.4.5 CODE REVIEWS FOR MALICIOUS LOGIC

It is a straightforward extension to perform code reviews specifically to discover malicious logic. This review process should be done by teams in order to ensure its validity. Reviews are conducted during the development process rather than afterwards by penetration teams.

These last two techniques (i.e., configuration control and code reviews) can go a long way to prevent insertion of malicious logic into a computer system. It is this author's opinion that they should always be a part of the selected off-line techniques.

2.5 TRADEOFFS

It is frequently difficult to satisfy all the desired objectives of a system (e.g., performance, security, fault tolerance, compactness, etc.). Since resources are always limited, it is essential to decide from the onset of a project the amount to dedicate to security concerns. Resources are both computer resources, such as millions of instructions per second, and project resources, such as a budget to perform verification.

To determine the amount of resources to dedicate, an acceptable degree of risk from the threats posed by malicious logic must be defined. Tradeoffs should be performed between security and other desired system objectives using a defined and acceptable degree of risk as a control on the investment in security mechanisms. Of course, in order to perform these tradeoffs, some idea of the costs and effectiveness of the proposed security mechanisms must exist.

Additionally, an analysis of the proportion of off-line versus on-line techniques to be used in the total security budget should be performed. Decisions of this type can be made based on the cost, effectiveness, and performance impact of each approach. For example, NVP can be very expensive in development and maintenance. Therefore, widespread use of this technique in certain software systems may be unlikely. Instead, it could be used selectively, as determined from a tradeoff study (see definition of selective redundancy in the glossary of fault tolerance terms).

Obviously, off-line techniques have the advantages of not affecting performance, weight, or power (i.e., attributes of the physical computer system). However, on large software programs or hardware components, their effectiveness may be too limited. This is evident from experiences with current formal verification technology.

CHAPTER 3

FAULT-TOLERANT, SECURE COMPUTING SYSTEMS

3.1 FAULT, ERROR, AND FAILURE IN SECURE COMPUTER SYSTEMS

The closest term to fault in computer security jargon is "flaw." This is defined as "An error of commission, omission, or oversight in a system that allows protection mechanisms to be bypassed" [DoD85a]. Clearly, this only includes design faults and not physical faults that can manifest themselves in computer hardware. Current computer security requirements and technology only deal with physical faults in a very superficial way. The following two excerpts from [DoD85a], under operational assurances, is the only treatment of this issue:

> System integrity - hardware and/or software features shall be provided that can be used to periodically validate the correct operation of the on-site hardware and firmware elements of the TCB.
>
> Trusted recovery - procedures and/or mechanisms shall be provided to assure that after an ADP [Automatic Data Processing] system failure or other discontinuity, recovery without a protection compromise is obtained.

System integrity is a requirement for secure computer systems throughout the class range of C1 to A1, while a trusted recovery requirement is only necessary for classes B3 and A1.

The problems with this limited treatment of physical faults are as follows: (a) system integrity has typically been interpreted as built-in-test on power up and reset, (b) such diagnostics are ineffective in detecting transient, and not very effective for intermittent permanent faults, and (c) trusted recovery has typically been interpreted as a secure reboot, not the recovery from the effects of errors. Even current research in hardware verification [Bevi87] addresses correctness only in the absence of faults.

An error in the fault tolerance sense, that results in compromise or unauthorized modification, cannot occur in secure computer systems currently in use or being built. Such a security relevant error would be a violation of a computer system's security policy (i.e., the specification of its security requirements), and would thus be a system failure! In secure systems, an error is immediately a failure whether or not a user has observed improper service at a service boundary. This is true by definition and use of security policies such as mandatory and discretionary access control [DoD85a]. (Note, that we are only addressing unauthorized disclosure of information and/or modification of information or programs.)

An example should make the above clear. Let two files exist in a secure computer system, file A at Top Secret, and file B at Secret. Lets suppose that due to some fault (physical fault in hardware, hardware or software design fault, accidental or deliberate) some of the Top Secret information from file A ends up in file B at Secret (thus a violation of the *-property [DoD85a]). Whether or not any Secret user has read the Top Secret data in file B or not, this error is immediately a failure of the security mechanisms of the system.

As discussed in [Turn86], the only fault tolerance technique that seems to be compatible with the security perspective discussed above is fault masking. This way the error never occurs, in the security sense, as long as the masking mechanism is not overwhelmed by errors. Clearly, another way to express this absence of errors in secure systems is that computer security techniques require fault avoidance. As soon as a fault manifests itself into a security relevant error a secure system will fail. Yet, another way to express this as done in [Turn86], is that security relevant errors are likely to be unrecoverable.

An exception to the above concerns is the unauthorized modification of data or programs in which there is a unharmed backup copy. For example, modification of network packets or data packets sent to a device. When the modification is detected (e.g., a cryptochecksum is invalid) it can be recovered from by retransmitting a copy of the original packet. Additionally, unauthorized modification of programs can be detected in the same way and recovered by simply obtaining another copy from a backup on disk. If a ROM with no backup is damaged by some deliberate hardware fault, then this will cause an immediate failure. Likewise, any modification of data values in a secure computer system seem likely to eventually lead to failure of its security mechanisms.

[Turn86] suggests a less strict interpretation of the standard application of security doctrine. A security relevant error will turn into a failure only if a compromise occurs. For example, in our file example above, the Top Secret data in file B (remember this is due to some fault) is only a failure of the secure system when a Secret user reads that data in file B, thus compromising it. This is closer to the fault tolerance notion of error, and thus, allows the application of other fault tolerance techniques than just fault masking.

To allow this approach, extensions to the accepted security policies and models of today (e.g., Bell–LaPadula Model) are needed (see Section 6.4.2). This seems inevitable for secure systems to provide proper enforcement of security policies in the presence of all types of faults with other than fault masking. Whereas, fault masking may be effective, total dependence on it can be very costly.

Specifically, current computer security technology relies on fault avoidance for the proper construction of a trusted computing base (TCB) [Dobs86], which is composed of both hardware and software. The trusted computing base encompasses all of the security mechanisms of a computing system. Its trusted components (e.g., security kernel, trusted processes (subjects) [Bell75] [GC84] [DoD85a]) are relied upon to provide proper service with respect to a system's security policy. This trust is validated via fault avoidance technology, which includes: formal specifications, software and hardware verification, testing, and code reviews. If a design fault (accidental or deliberate) exists in the TCB of a deployed system, then that system will likely fail since the TCB is void of any fault tolerance mechanisms. An example, of the lack of fault tolerance mechanisms in security relevant hardware, is the hardware implementation of a reference monitor [DoD85a] in the Secure Ada Target (SAT) [Boeb85b] [Boeb85c] (this is now called "LOCK" [Sayd87], and will be called this throughout the remainder of this dissertation).

Secure systems tend to have more self-checks, or acceptance tests, than normal computer systems. These acceptance tests are designed to enforce a security policy, and thus, scrutinize the actions of less trusted code, but not the actions of the TCB itself. These extra acceptance tests are likely to catch several errors resulting from both hardware and software faults that regular systems would fail on. However, by no means do these tests come close to providing the fault tolerance desired. The faults caught in this way will likely be mistaken, for at least a short time, as security violation attempts. One such example of this, are the restrictions placed on information flow in the Bell and

LaPadula model [Bell75]. When this is implemented in a secure system any fault, which causes or aids a subject to violate this security policy, may be caught by one of the numerous acceptance tests that enforce it. (Notice, that such restrictions on information flow set up error containment barriers to prevent the leakage of information.) However, since these defenses where not designed to cope with physical and design faults some of these security violations are likely to go unnoticed.

Additionally, security requirements are directed at ensuring that the TCB is functioning correctly. If faults occur that leave the TCB functioning correctly, then they are of no concern to the security mechanisms of a system. Only faults that can compromise the TCB, directly or indirectly, are important. Now, if we enlarge our security concerns to include denial–of–service and integrity of function and data the above restriction changes. The rest of a computer system (i.e., even outside the TCB) is now important in order to provide proper service.

3.2 FAULT, ERROR, AND FAILURE CLASSIFICATION OF SECURITY THREATS

Fault Environments for a Secure System

The entire life–cycle of a secure computer system must be covered [Myer80]:

1. Development: requirements, specifications, algorithm design, hardware and software design, and implementation.

2. Pre–deployment: engineering changes, maintenance.

3. Deployment: maintenance (hardware, firmware, and software), and new releases.

Fault Classes for a Secure System

1. Malicious Logic—deliberate faults: computer viruses, Trojan horses, trap doors, software design faults, hardware design faults, algorithm faults, and specification faults (the first 3 cases are design faults, which have specific names assigned to them in the security literature).

2. Accidental—nondeliberate faults: (i.e., classical security concerns –flaws) software design faults, hardware design faults, algorithm faults (e.g., crypto algorithm not adequate), specification faults, requirements faults (e.g., in the last 2 cases, insufficient security policy and/or models), and physical faults.

3. Covert Channels—deliberate, accidental, or unavoidable faults; these can be thought of as breaks in the error isolation boundaries of a multilevel secure system (i.e., imperfect coverage of the security mechanisms): storage covert channels, timing covert channels, and tempest (i.e., the study and control of spurious electronic signals emitted from ADP equipment [DoD85a]).

4. Interference—deliberate faults: radiation, physical attack, overloading input, and jamming.

5. Interaction—deliberate or accidental faults: hardware or software maintenance (e.g., insertion of malicious logic), and operational (e.g., improper system initialization, see Figure 3.8). This fault class describes the situation where an authorized, trusted user (i.e., human) performs actions against a defined security policy, and is termed the "Insider Threat" [Clyd87].

Error Classes for a Secure System

1. Unauthorized disclosure of sensitive information, examples: (1) accessing data in a file system, computer's state (e.g., register file), or data base, (2)

intercepting authorized communication in transmission channels, (3) having authorized access but performing unauthorized dissemination, (4) unauthorized execution access to programs [Glig83], and (5) issues in database security, such as, inference [Denn82], aggregation, and penetration [Henn86].

2. Invalid modification of objects:

Unauthorized modification of information, programs,[1] and hardware (includes firmware) (i.e., these are classical integrity concerns [Port85]), (1) making changes to data values or programs [Shir81], (2) intercepting communications, making changes, and then retransmitting [Voyd83], (3) loading data or programs into the system during run time, and (4) computer viruses errors.

Preventing fraud and errors [Clar87], (1) a user must be prevented from manipulating data in an arbitrary manner, even if given modify access to it, and (2) a user must be prevented from performing all subactions of a well defined transaction for fear that he/she will do so to commit fraud (see Section 2.2.2). Manipulations must be constrained in ways that preserve internal consistency of data, since multiple data items may be used to derive the state of a real world object.

Note, that if either unauthorized modification or preventing fraud and errors are extended from a user to a subject (i.e., include software and/or hardware actions), then there is overlap of this error class and class 3 (see Figure 3.15 b).

3. Integrity of function and data:

Integrity of function[2] (i.e., proper service or design correctness, how correct the functioning of a process is [Port85], software and/or hardware

[1]Includes database queries before invocation.

authorized to perform a specified function, but maliciously provides incorrect service—authorized misuse), (1) performing the wrong functions, performing functions the wrong number of times, and/or in the wrong order, (2) performing actions that a function was not specified for (i.e., added functions), (3) omitting functions, lack of completeness, (4) generating incorrect results,[3] (5) providing erroneous input to a function (e.g., switch order of parameters), (6) producing outputs without inputs (i.e., arbitrarily) [Ezhi86], (7) producing outputs too early or too late, and (8) performing functions during the wrong real world time (e.g., wrong time of day, wrong day in the week, wrong quarter of the year).

Integrity of data (i.e., how correct information in an object is [Port85]), (1) a subject (i.e., just hardware and software subjects) authorized to modify a data item, but maliciously does so resulting in an incorrect data item [Shan77, p.61] (e.g., using old data), (2) data from the wrong source, improper authentication, (3) structural integrity of data structures (i.e., the correct manipulation and state of a data structure, for example, the incorrect removal of an entry in a linked list) [Tayl80], (4) accuracy, precision, completeness, timeliness, and consistency, and (5) obviously, loss of integrity of function can result in loss of

---

[2]An interesting instance of a possible malicious attack to cause a loss of integrity of function in databases appears in [Henn86]. The integrity of database queries have several facets: (a) proper query interpretation, the requested information is what is actually returned (i.e., loss of data, wrong data), and (b) malicious logic insertion into a query when it is translated into database actions. In case "a," the design correctness of the database operations are in doubt, and in case "b," the correctness of the translator is in question.

[3]For example, false alarm indication from hardware and/or software error detection mechanisms.

integrity of data.

4. Denial-of-Service: is when "a group of authorized users of a specified service is said to deny service to another group of authorized users if the former group makes the specified service unavailable to the later group for a period of time which exceeds the intended (and advertised) service MWT" (Maximum Waiting Time) [Glig83]. That is, denial-of-service is a lack of guaranteed access to a shared service.

"Guaranteed access ensures that authorized users cannot prevent any other authorized users from accessing shared data, programs, and hardware resources. ... In more general terms, no authorized user is able to deny the access of any other authorized user to a shared service" [Glig83].

The above definitions, and a scheme for detecting denial-of-service based on setting a watchdog timer [Glig83], imply that another authorized user does not actually have to be waiting for the service for denial to occur. Just the fact that a service is inaccessible is sufficient.

In [Glig85a], it is presented that loss of the integrity of data results in denial-of-service (access) to that data. In [Glig83] [Glig85a], many examples of denial-of-service not caused by integrity are given (e.g., incompatible or inadequate resource control algorithms and policies). Thus, not all denial-of-service instances are caused by loss of integrity. Also, [Glig85a] defines denial-of-service as occurring only when services are shared. However, we extend this to include denial of specified behavior of any software and/or hardware (i.e., loss of integrity of function, and thus, is integrity-based). We call this "insider-generated" denial-of-service, since it is caused by deliberate design faults, and requires no interuser dependency. In fact, [Glig83] includes a "misbehaved service" as a denial-of-service, because the specified service is unavailable, even if it performs in the MWT. However, he limits it to the case of a service being modified by an intruder (i.e., another instance of intruder-generated denial-of-service), that takes advantage of some flaw in the access mechanisms or policies (e.g., computer viruses). In [Neum78], denial-of-service is defined as maintaining the integrity of a resource.

5. Spoofing: (1) trick a system into doing something it should not do (e.g., enemy gives the system commands or status information), and (2) trick user to believe that the enemies' system is the real system.

6. Learning a system's capabilities: in SDI, the Russians learn through small engagements what the system can and cannot do.

7. Traffic analysis of networks.

Fault and Error Relationships

A fault appearing on the left hand side implies, that at least one particular instance of a fault from that class can cause that error, NOT necessarily that all can.

FAULT CLASSES:	ERROR:
Malicious Logic Covert Channels Accidental (e.g., flaw) Interaction (e.g., operational)	Unauthorized disclosure of sensitive information
Malicious Logic (e.g., trap door, virus) Accidental Interaction	Unauthorized modification of information, programs, and hardware
Malicious Logic Accidental Interaction	Integrity of function and data

Malicious Logic Accidental Interaction Interference	Denial-of-Service
Malicious Logic Accidental	Spoofing
Accidental Interaction	Learning a system's capabilities
Malicious Logic Accidental	Traffic analysis of networks

The rest of this section presents the remainder of the fault, error, and failure classification by the use of a standard fault and error classification procedure [Aviz87a], and by the use of fault trees. Tables 3.1 and 3.2 contain the standard classification with its key provided below. Figure 3.1 presents a graphical characterization of a computer virus fault and error. Figure 3.2 to 3.13 present the fault trees for all the security threats presented above. Figure 3.14 and 3.15 describe the relationships between errors.

KEY: Elements of a Fault Classification [Aviz87a]

By Count:	Single(S) Versus Multiple(M)
By Origin:	Physical(P) Versus Human-Made(H)
By Activity:	Dormant(D) Versus Active(A)
By Intent:	Accidental(A) Versus Deliberate(D)
By Duration (of activity):	Transient(T) Versus Permanent(P) (or Intermittent(I))
By Extent:	Local(L) Versus Distributed(D)
By Value:	Fixed(F) Versus Variable(V)
By Time (multiple):	Coincident(C) Versus Separated(S)

By Consistency:	Time(T) Versus Value(V)
By Cause (multiple):	Independent(I) Versus Related(R)

KEY: Elements of an Error and Failure Classification

ERRORS:	
By Count:	Single(S) Versus Multiple(M)
By Manifestation:	Latent(L) Versus Detected(D)
By Form:	Identical(I) Versus Similar(S) Versus Distinct(D)
By Cause:	Independent(I) Versus Common(C)
By Nature:	Value(V) Versus Time(T) Versus Consistency(C)
FAILURES:	
By Consequence:	Ordinary Versus Catastrophic (Benign Versus Malign)

Faults by:	Count	Origin	Activity	Intent	Duration	Extent
<u>Malicious Logic</u>						
Computer Virus	M	H/P[1]	D	D	P&I[2]	D
Trojan Horse	S/M	H	D	D	P&I	L/D
Trap Door	S/M	H	D	D	P	D
Software Design Faults	S/M	H	D	D	P&I	L/D
Hardware Design Faults	S/M	H	D	D	P&I	L/D
Specification	S/M	H	A	D	P	L/D
Algorithm	S/M	H	A/D	D	P&I	L/D
<u>Covert Channel</u>						
Storage	S/M	H	D	A/D	P/I	L
Timing	S/M	H	D	A/D	P/I	L
Tempest	S/M	H/P	D	A/D	P/I	D
<u>Interaction</u>						
Maintenance	S/M	H/P	D	A/D	P	L/D
Operational	S/M	H	A	A/D	T	L/D
<u>Interference</u>						
Radiation	M	H/P	A	A/D	T	D
Physical Attack	S	H	A	D	T	D
Overload	S	H/P	A	A/D	T	D
<u>Accidental</u>						
Software Design Faults	S/M	H	D	A	P&I	L/D
Hardware Design Faults	S/M	H	D	A	P&I	L/D
Specification	S/M	H	A	A	P	L/D
Algorithm	S/M	H	A/D	A	P&I	L/D
Requirements	S/M	H	A	A	P	D
Physical Faults	S/M	P	A/D	A	P,I,T	L/D

Table 3.1 Fault Classification of Security Threats

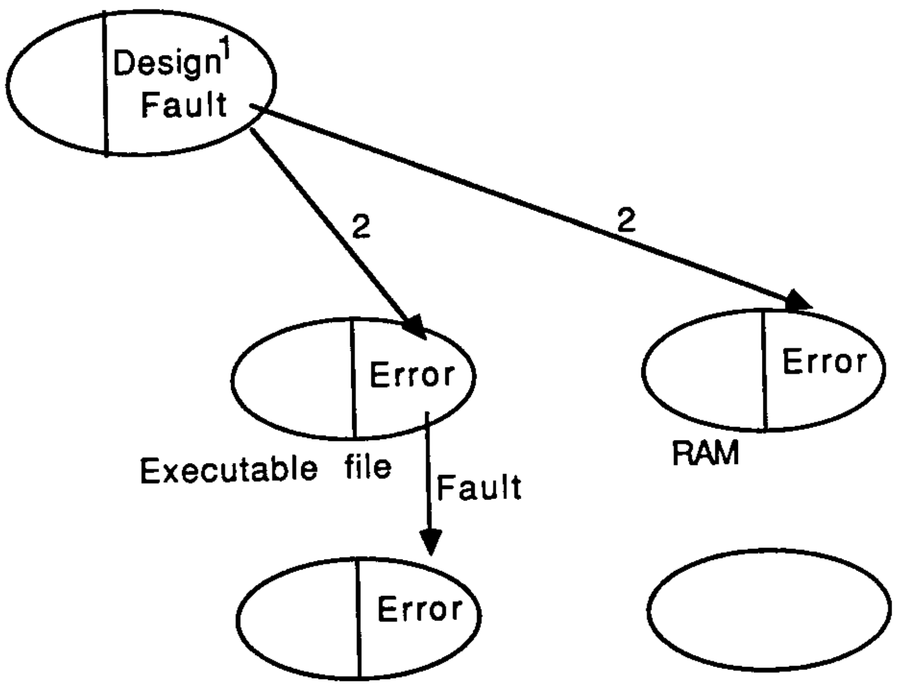
Faults by:	Value	Time	Consistency	Cause
<u>Malicious Logic</u>				
Computer Virus	F/V[4]	S[5]	T[4]	R
Trojan Horse	F/V	S	T[6]	R/I[3]
Trap Door	V	S	?	R
Software Design Faults	F/V	S/C	?	R
Hardware Design Faults	F/V	S/C	?	R
Specification	F/V	C	T	R
Algorithm	F/V	S/C	?	R
<u>Covert Channel</u>				
Storage	V	S(/C)	V[7]	R/I
Timing	V	S(/C)	V	R/I
Tempest	V	--	V	--
<u>Interaction</u>				
Maintenance	F/V	S/C	?	R
Operational	F/V	S/C	?	R
<u>Interference</u>				
Radiation	F/V	C	?	R
Physical Attack	F/V	--	?	--
Overload	F/V	--	V	--
<u>Accidental</u>				
Software Design Faults	F/V	S/C	?	R/I
Hardware Design Faults	F/V	S/C	?	R/I
Specification	F/V	C	?	R
Algorithm	F/V	S/C	?	R
Requirements	F/V	C	?	R
Physical faults	F/V	S/C	?	I

Table 3.1 Fault Classification of Security Threats (continued)

Errors by:	Count	Manifestation	Form	Cause	Nature	Failures by: Consequence[10]
Disclosure	M	L/D	S	I/C	V	Depends on data
Modification	S/M	L/D[8]	S	I/C	V,C	Depends on changes
Computer Virus	M[9]	L/D	I/S	C	V	Usually Catastrophic
Integrity	S/M	L	S	I/C	V,T,C	Depends on changes
Denial-of-Service	S/M	L	S	I/C	V,T,C	Usually Catastrophic
Learn a system's capabilities	S	L	?	I/C	V,T	?
Spoofing	S	L	S	I/C	V,T,C	Depends on action
Traffic Analysis	S/M	L	S	I/C	V,T	?

Table 3.2 Error and Failure Classification of Security Threats

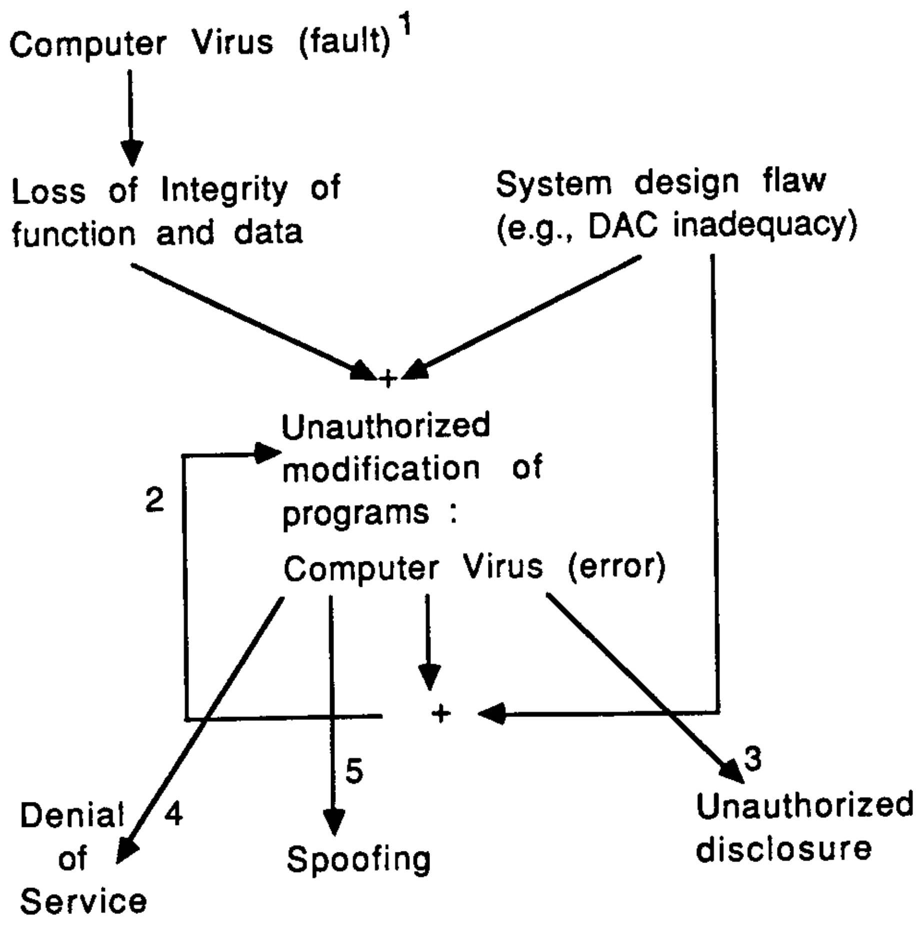
Footnotes for Tables 3.1 & 3.2:
1: ARPANET case [Rose81] [Neum86].
2: Likely to be intermittent.
3: The same or different people responsible.
4: A virus can evolve.
5: Takes time to spread.
6: Unauthorized disclosure or computer virus.
7: Unauthorized disclosure.
8: e.g., integrity-lock in databases.
9: Propagate to all executables.
10: Degrees of catastrophe exist, e.g., release of Top Secret data is more severe than Secret data.



fault → error → fault → error
The computer virus error propagates.

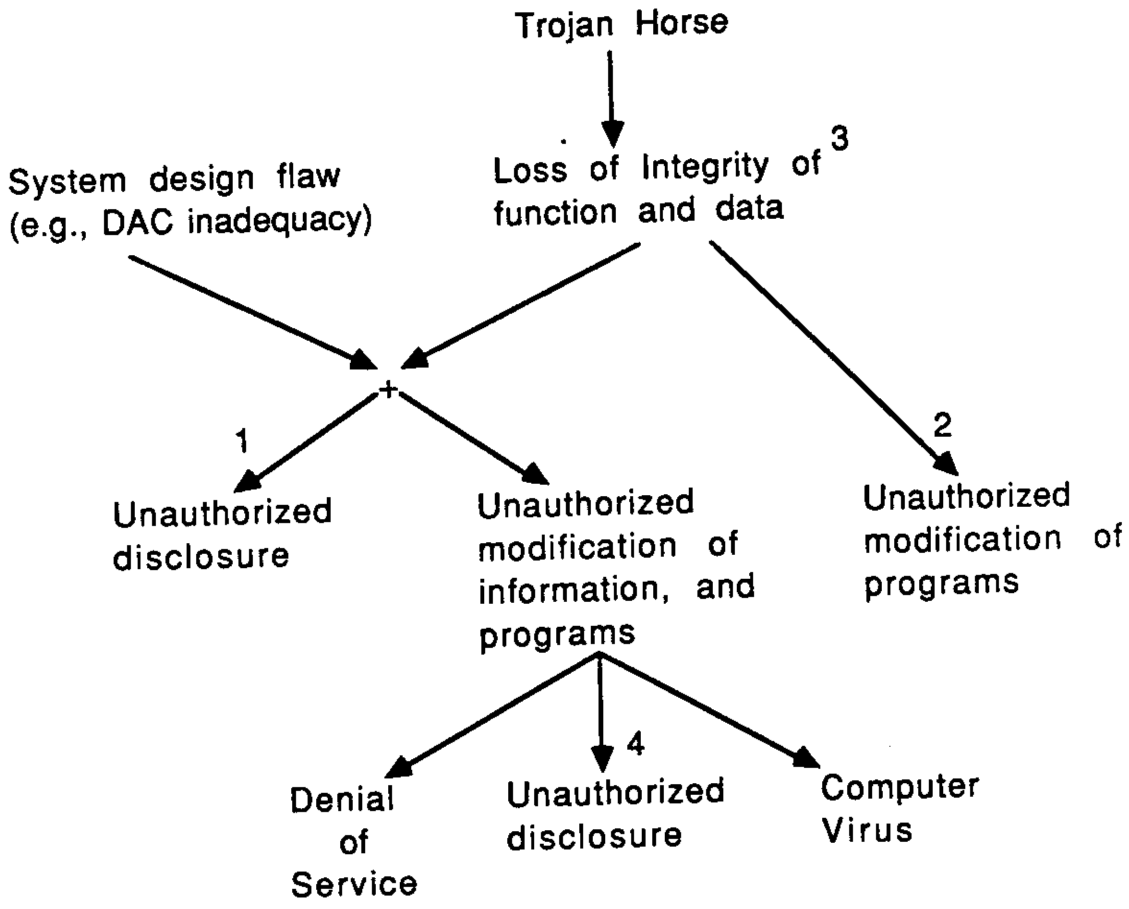
1: Initially a computer virus can be a special Trojan horse that injects the virus into a computer system [Pozz86]. This is a deliberate design fault.
2: e.g., the virus writes to an executable file, or unprotected part of RAM such as a process's stack space in the Intel 8086 processor.

Figure 3.1 Computer Virus: A Fault and an Error


Computer Virus (fault) 1
Loss of Integrity of function and data
System design flaw (e.g., DAC inadequacy)
+
Unauthorized modification of programs :
Computer Virus (error)
2
+
3
5
4
Denial of Service
Spoofing
Unauthorized disclosure

1: Initially a computer virus can be a special Trojan horse that injects the virus into a computer system. This is a deliberate design fault.
2: Infection property (i.e., error propagation), or loading a program into a privileged domain (i.e., gain of privileges).
3: e.g., insert use of a covert channel.
4: [Glig83, p.140]: "...it is possible that a malicious user can modify the intended service behaviour in a non-obvious way by exploiting design flaws in the service access mechanism or policy. ...misbehaved service.." [Cohe84]: place all infected executables into an infinite loop, thus resulting in CPU resource denied.
5: e.g., computer virus runs before original program, and pretends to be the original program.

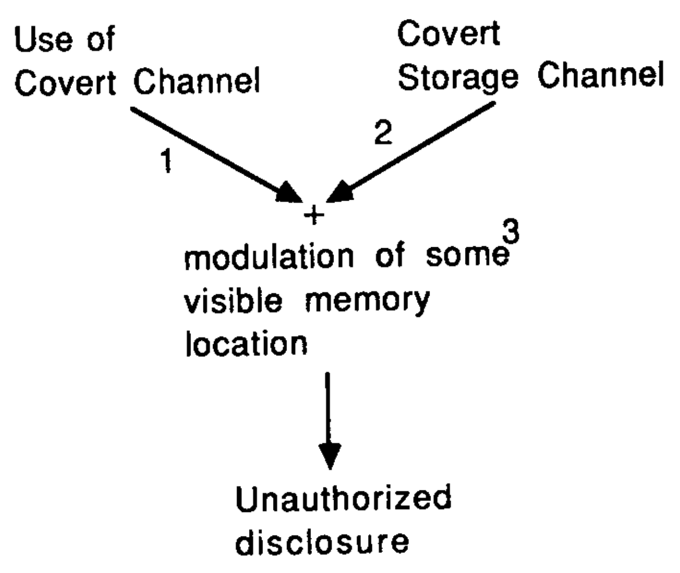
Figure 3.2 Fault Tree for a Computer Virus



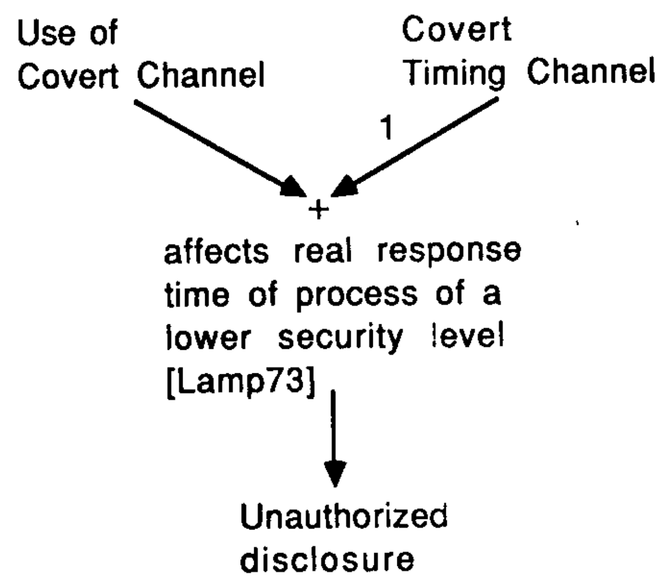
Trojan Horse
System design flaw (e.g., DAC inadequacy)
Loss of Integrity of function and data
3
+
1
Unauthorized disclosure
Unauthorized modification of information, and programs
2
Unauthorized modification of programs
Denial of Service
4
Unauthorized disclosure
Computer Virus

1: e.g., dissemenation via covert channel [Lamp73] [Scha77] [Loep85].
2: Add functionality which does not cause denial-of-service directly, e.g., trap door insertion [Thom84].
3: Includes unexpected and malicious side effects [Denn82].
4: e.g., a Trojan horse can change the discretionary access rights for all (or part) of the subject's objects, so that the Trojan horse's owner can access then at any time [Down85].

Figure 3.3 Fault Tree for a Trojan Horse

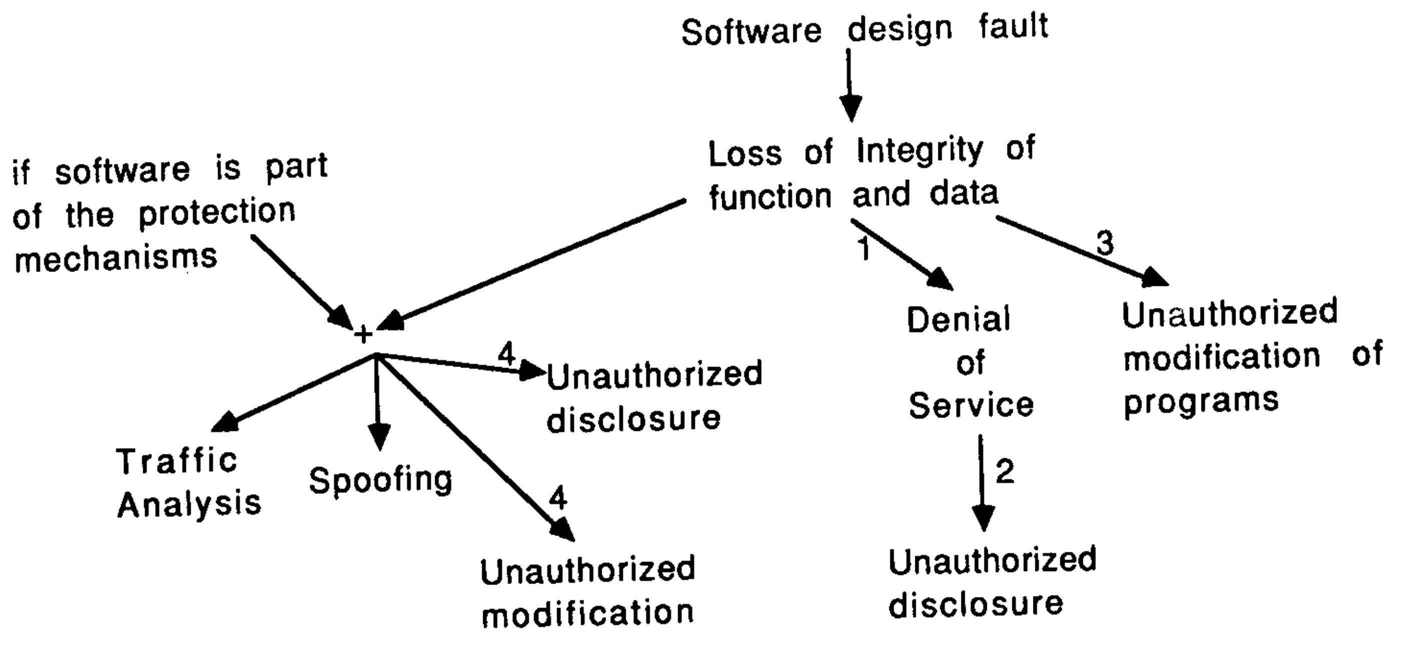


1: Deliberate use of a covert channel is a deliberate design fault, may be prevented by N-Version Programming [Dobs86].
2: The existance of this can be an accidental or deliberate algorithm fault. Can also be unavoidable due to resource sharing, thus this is NOT a fault.
3: Direct/indirect writing by high level subject, and direct/indirect reading by lower level subject.



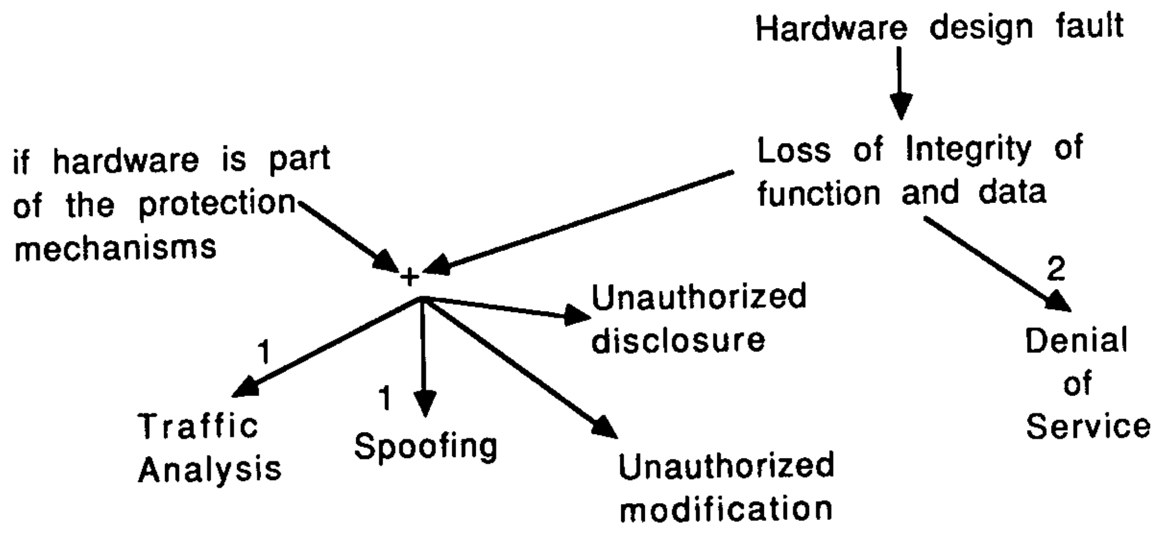
1: Deliberate or accidental algorthm fault, e.g., bad resource sharing mechanisms--page fault example in [Lamp73] [Scha77] [Loep85].

Figure 3.4 Fault Trees for Storage and Timing Channels


Software design fault
Loss of Integrity of function and data
if software is part of the protection mechanisms
+
1
2
3
4
4
Denial of Service
Unauthorized modification of programs
Unauthorized disclosure
Traffic Analysis
Spoofing
Unauthorized disclosure
Unauthorized modification

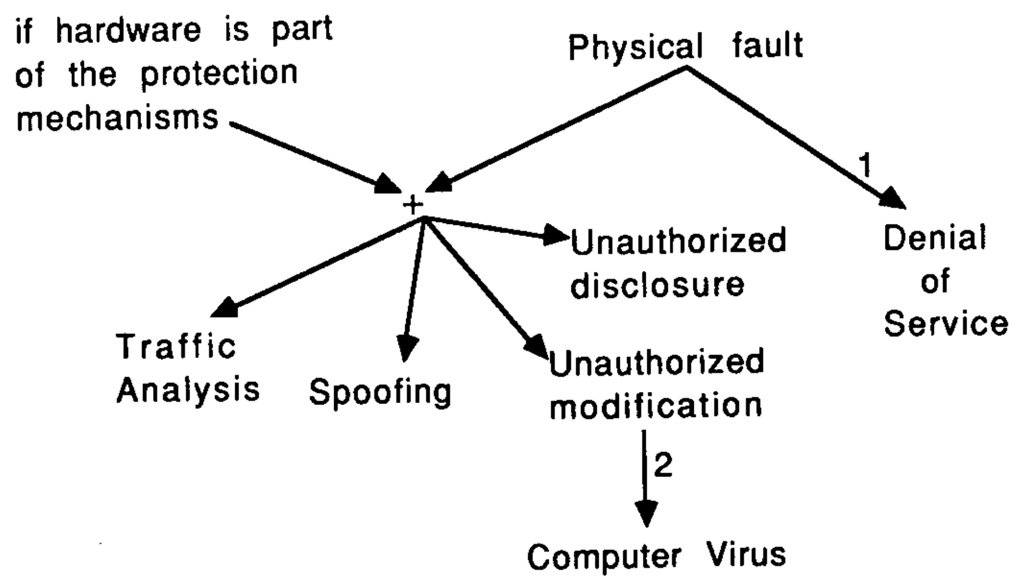
1: A software function is a service, can cause denial-of-service by delaying the response or by providing improper service. NVP will force a correct response in a specified period of time, because of forward error recovery.
2: Covert channel by resource denial [Glig83].
3: Unauthorized modification by program handling tools [Thom84].
4: Can change the security decision support data (e.g., labels, access rights) [Turn86], thus this case falls under the broad definition of a trap door [DoD85].

Figure 3.5 Fault Tree for Software Design Faults


Hardware design fault
Loss of Integrity of function and data
if hardware is part of the protection mechanisms
+
Unauthorized disclosure
1
Traffic Analysis
1
Spoofing
Unauthorized modification
2
Denial of Service

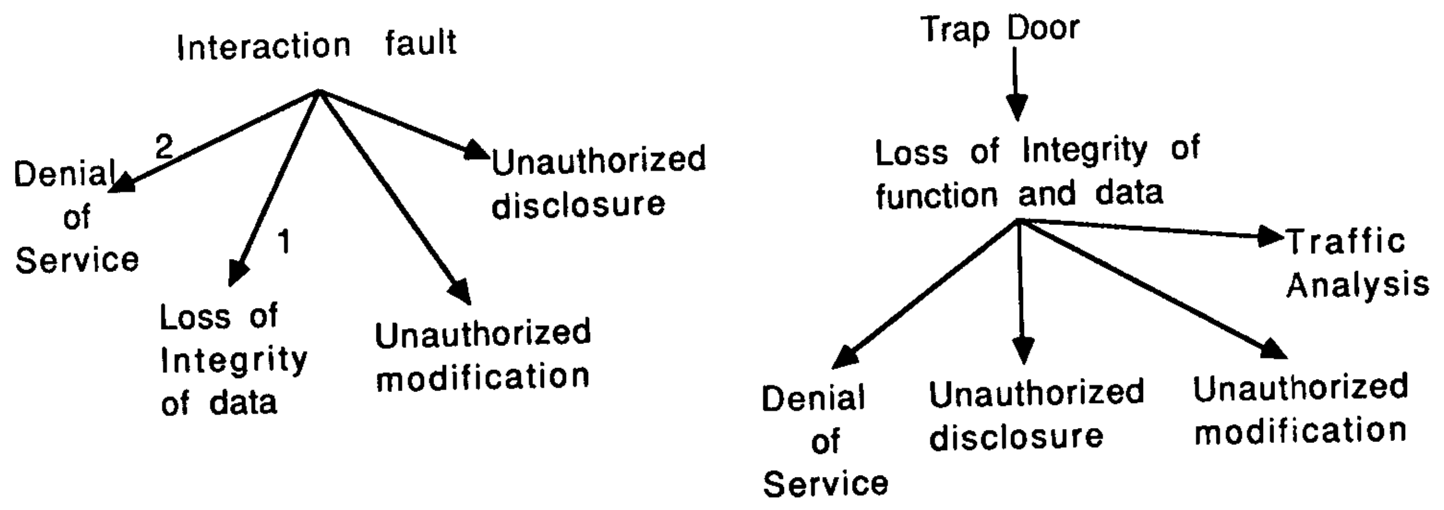
1: e.g., design fault in encryption hardware generates plain text or easily boken code (receiving or sending side should detect this).
2: A hardware function is a service, can cause denial-of-service by delaying the response, by providing improper service, or by degrading in performance and in functionality, e.g., may lose some hardware needed for security.

Figure 3.6 Fault Tree for Hardware Design Faults



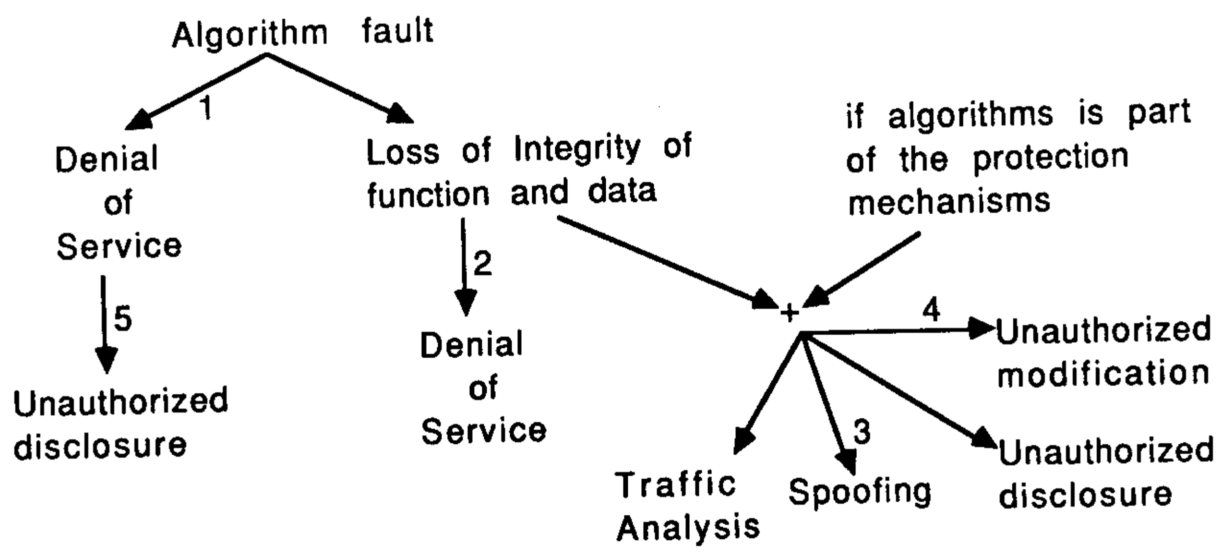
1: Degradation of hardware components results in the loss of application and system support functionality (e.g., hardware and software for fault tolerance and computer security).
2: ARPANET case of a hardware fault generated computer virus [Rose81] [Neum86].

Figure 3.7 Fault Tree for Physical Faults



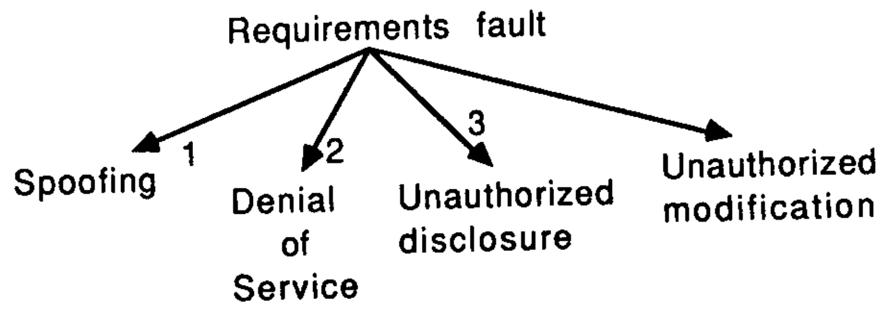
1: Trap door insertion via improper initial state. Authorized user makes invalid changes by adding improper access rights for new or old subject (e.g., the unauthorized password "Joshua" in the movie Wargames [Land85]).
2: Remove some system resource from general use, e.g., editors. In UNIX change a users search path so that an imposter program is used. This results in the denial of the proper routine(s). Could also run a system in a degraded mode on purpose (e.g., no acess control center [Fell87]).
*: All arcs can be attributed to an authorized trusted user or designer performing actions against the defined security policy. This is termed the "Insider Threat" [Clyd87].

Figure 3.8 Fault Trees for Interaction Faults and Trap Doors



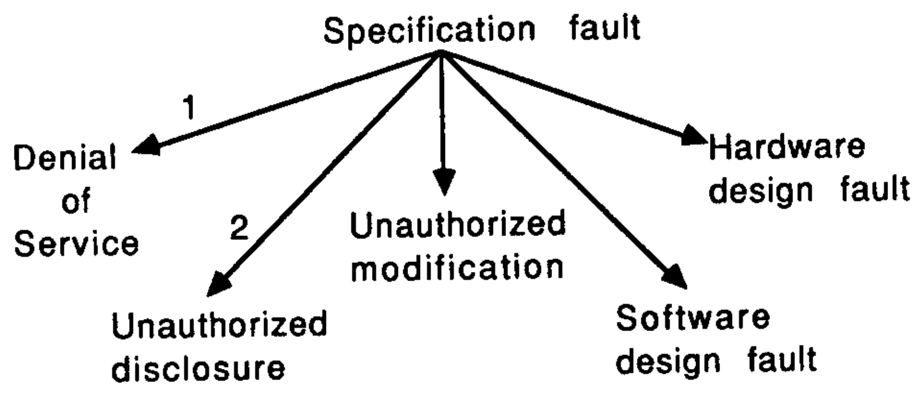
1: Service-generated denial-of-service caused by undesired interuser dependencies, or intruder-generated denial-of-service.
2: Specified functions not performing as required.
3: e.g., weak encryption algorithm, or incorrectly accepts authentication.
4: e.g., weak encryption allows successful network packet modification without detection.
5: Denied access to an object can be used as a covert channel [Glig83].

Figure 3.9 Fault Tree for Algorithm Faults



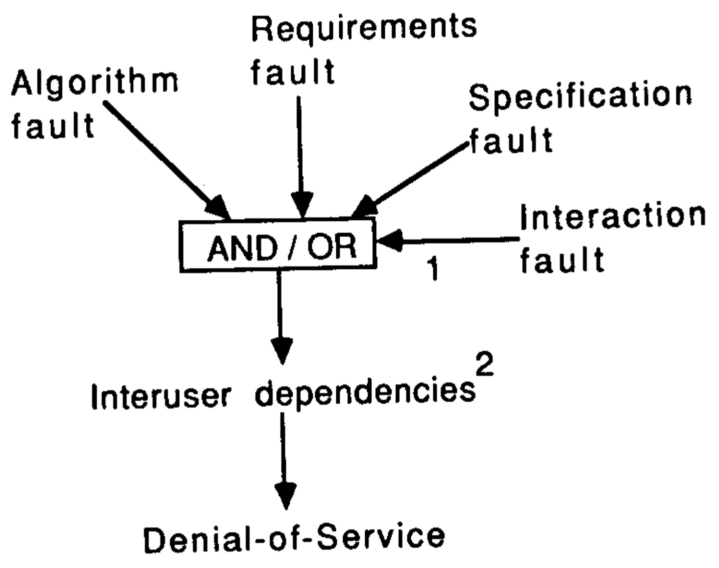
1: e.g., absence of a trusted path can lead to the classical example of spoofing. A program pretends to be the computer system login program that asks for the user's password. Once it obtains the users password it fakes a system malfunction and transfers control to the real system. The user has no idea of what happened.
2: Improper combinations of service sharing policies, incorrect or improper combinations of deadlock handling policies, and concurrency control policies [Glig83] [Glig85a].
3: Insufficient security policy and/or incomplete set of security threats.

Figure 3.10 Fault Tree for Requirements Faults



1: Insufficient setting of maximum waiting time (MWT) [Glig83], or not specifing a MWT for a service at all.
2: Insufficient security model (i.e., model of how the security policy will be enforced).
*: All arcs can cause either an accidental or deliberate design fault.

Figure 3.11 Fault Tree for Specification Faults



1: Users dependent on trusted operator.
2: Can be undesired, desired, or unavoidable.

Figure 3.12 Non-Integrity Caused Denial-of-Service

Types of denial-of-serivce:
1) service-generated (e.g., inside service due to improper sharing policies or mechanisms),
2) intruder-generated (e.g., network packet modification) [Glig85a],
3) insider-generated (i.e., deliberate design fault results in integrity-based denial-of-service, and no interuser dependency is needed).

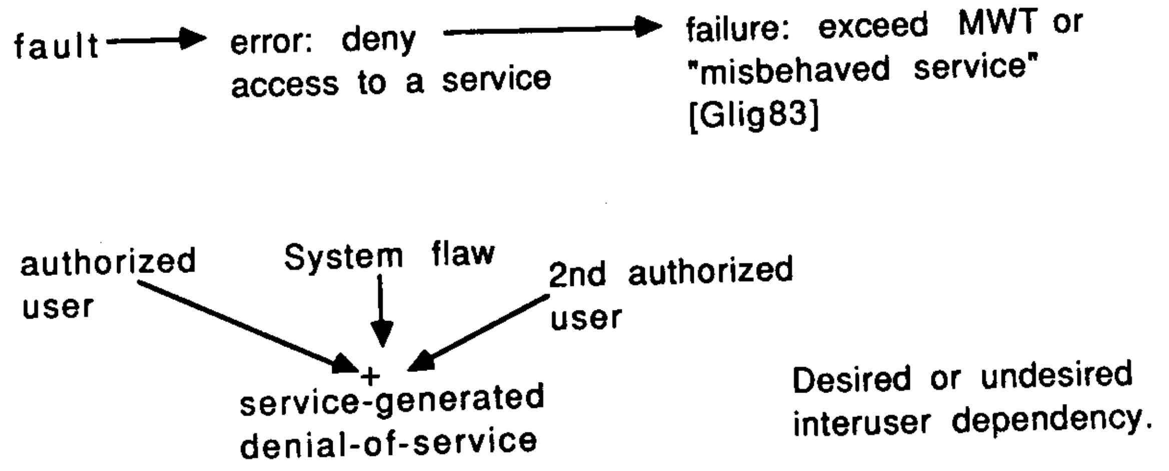

fault
error: deny access to a service
failure: exceed MWT or "misbehaved service" [Glig83]
authorized user
System flaw
2nd authorized user
+
service-generated denial-of-service
Desired or undesired interuser dependency.

Figure 3.13 Types of Denial-of-Service

Requirement Faults

#1 - inadequate sharing policies,
#2.3 - inadequate concurrency control (policies),
#3.2 - combinations of incompatible policies.

Algorithm Faults

#2 - inadequate sharing mechanisms,
#2.1 - inadequate bounds on resources (example b- bad mechanism design of processor sharing, example d- inadequate enforcement [circumvention] of resource bounds, and example f- inadequate handling of resource-bound exceptions),
#2.2 - inadequate access control mechanisms (example b- built-in system dependency on user's behavior [a good general example of inadequate access control is discretionary Trojan horses [Down85]]),
#2.3 - inadequate concurrency control (mechanisms),
#3 - combinations of seemingly adequate sharing mechanisms and policies,
#3.1 - combinations of mechanisms (example a- legitimate denial of service, example b- concurrency and recovery control), #3.2 - combinations of incompatible (mechanisms).

Design Faults

#2.1 - inadequate bounds on resources (examples a, b, c- deliberately improper use of sharing mechanism [i.e., authorized user is not specified to take all of the available resources [Jose87]], example e- inadequate handling of resource - bound exceptions),
#3.1 - combinations of mechanisms (example c- incompatible conventions).

Non-classified Cases

Examples "a" and "c" in section 2.2 - inadequate access control mechanisms, does not seem to fit in any fault class (example a- built-in user dependency on other users' behavior, and example c- built-in system dependency on user's behavior).

Table 3.3 Classifying denial-of-service examples from [Glig83] into fault classes. Each "#" above is an example from [Glig83, Appendix A].

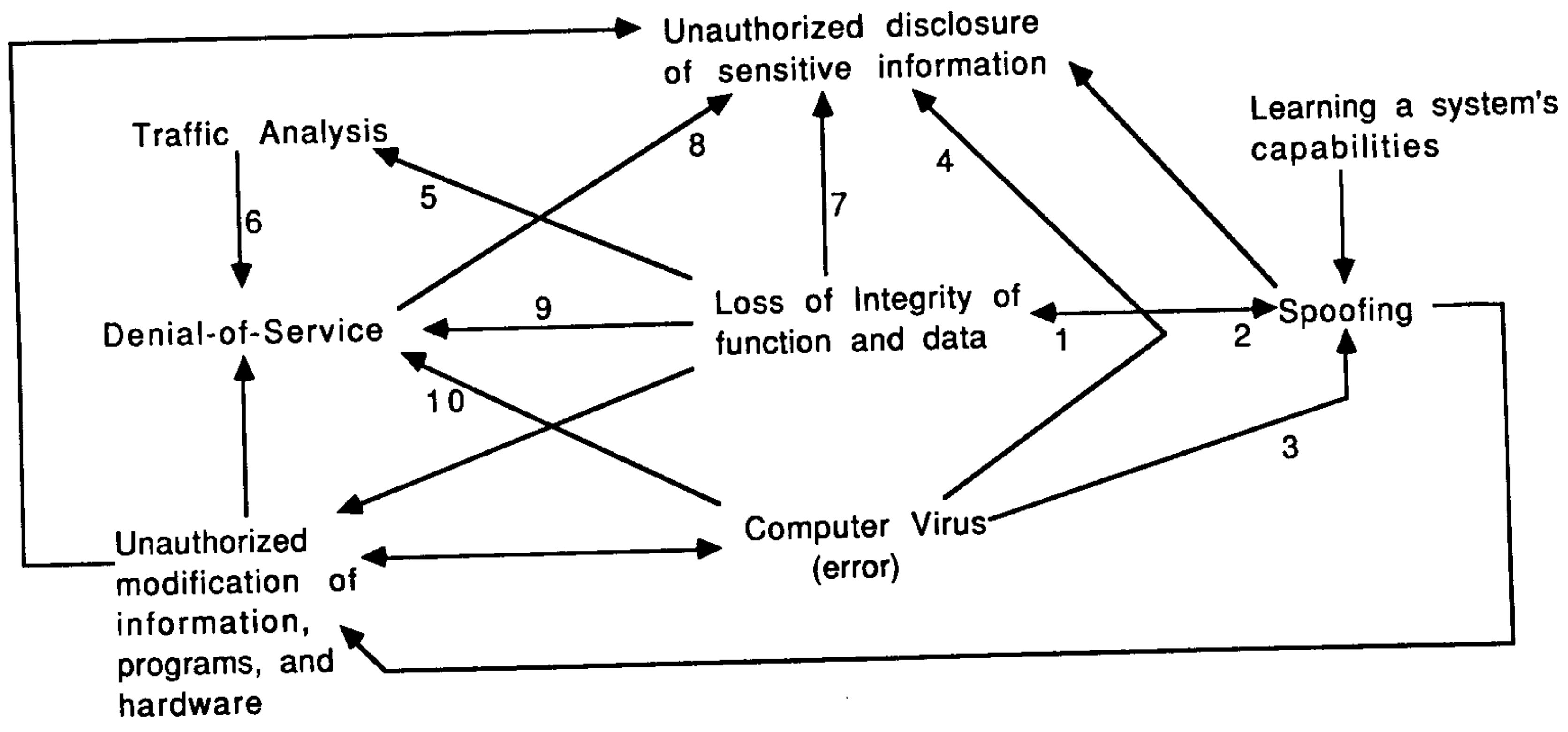

Unauthorized disclosure of sensitive information
Traffic Analysis
Learning a system's capabilities
Denial-of-Service
Loss of Integrity of function and data
Spoofing
Computer Virus (error)
Unauthorized modification of information, programs, and hardware
1
2
3
4
5
6
7
8
9
10

Figure 3.14 Possible Error Propagations

Key on next page ⟶

KEY: "$\longrightarrow$" can directly cause

1: e.g., cause loss of integrity by providing old data.

2: e.g., cause spoofing by incorrectly computing authentication.

3: e.g., computer virus runs before original program, and pretends to be the original program.

4: e.g., inject use of covert channel.

5: e.g., software / hardware design fault in encryption mechanism.

6: Helps to determine key communication links and nodes to attack.

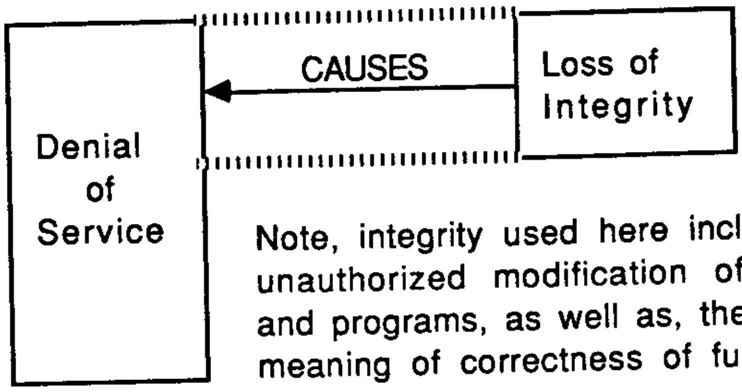
7: e.g., design fault in protection mechanisms (trap door), or use of a covert channel.

8: Covert channel (storage or timing) by resource denial [Glig83].

9: e.g., "misbehaved service," specified service is unavailable even if performed in MWT [Glig83]; extend to include incorrect behavior caused by deliberate design faults (i.e., insider-generated denial-of-service).

10: An additional example not in Figure 3.2, which pertains to both computer virus design faults and errors, is: denial-of-service due the repeated infection of the same executable(s).

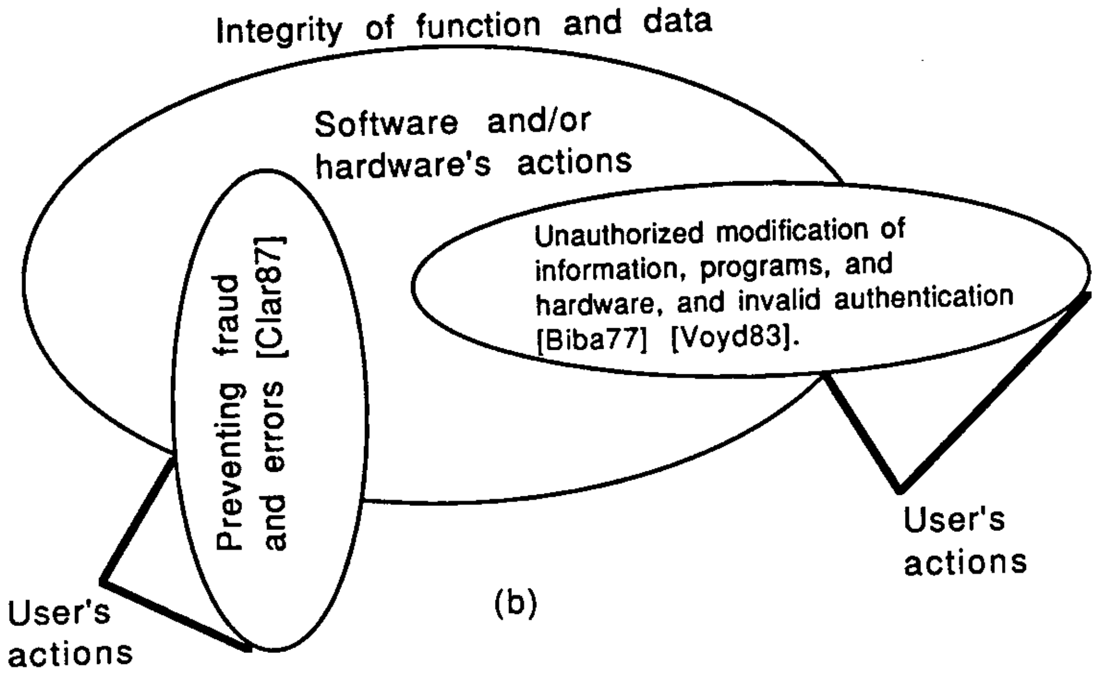
Figure 3.14 Possible Error Propagations (continued)



Note, integrity used here includes the unauthorized modification of information and programs, as well as, the broader meaning of correctness of function and data.

All cases of loss of integrity result in some form of denial-of-service, since some service is denied to a user. However, not all cases of denial-of-service are caused by loss of integrity [Glig85a].

(a)


Integrity of function and data
Software and/or hardware's actions
Preventing fraud and errors [Clar87]
Unauthorized modification of information, programs, and hardware, and invalid authentication [Biba77] [Voyd83].
User's actions
User's actions

(b)

Figure 3.15 Error -- Error Relationships

3.3 CRITICAL STATE FOR A SECURE COMPUTER SYSTEM

Before we start to devise detection and recovery schemes to handle both deliberate and accidental faults in a fault-tolerant, secure computer system, we must first define what its critical state is. This is the state that must be protected for an effective recovery to be possible. Once this last element of the characterization of faults and their effects is obtained then detection and recovery schemes can be devised.

The following definition of fault-tolerant security helps identify the critical state: "The security function is fault-tolerant if, despite of faults in the system [a reasonable set of faults to protect against must be defined, since protection against all faults is impossible], the security decisions correctly enforce the system's security policy, the associated decision support data (i.e., identifications, security labels, access rules) remain correct, no sensitive data are erroneously released, no covert channels are introduced, and no denial of service event takes place." [Turn86].

Critical state is composed of the following elements:

1. Encryption Keys [Denn82] [Wu87]: (a) data-encryption keys (likely to have different keys per security level of data), (b) key-encryption keys or the master key(s)—most (or groups) of other keys are encrypted in the master.

2. Access control data is used to enforce discretionary access controls [Denn82] [Down85]. Every object (e.g., can be a network connection) in the system has a list of subjects and their allowed access rights. Objects cannot be accessed (read/write/execute) without checking the defined access rights first.

3. Security level data is used to enforce mandatory access controls and is implemented by security labels. Every subject and object in a computer system must have one (i.e., the actual extent of this depends on the class of secure

system [DoD85a]). For a subject to access an object both must have valid labels.

4. Unauthorized modification control data is used to ensure only authorized modification of information and programs. Biba's integrity[4] levels [Biba77], and LOCK's type enforcement are possible implementations. Important aspects of this critical state are: (a) integrity levels are implemented by integrity labels [GC84], and (b) type enforcement is implemented with a type and domain indication for each object.[5]

5. Subject identification data is used to identify a user of the computer system for access control purposes, and audit of actions. Examples are, password files (i.e., authenticates a users authorization to use the system), and user login names.

6. Unmodified copy of TCB code is essential, any non-security relevant code is of lesser importance.

3.4 DETECTION, MASKING, AND RECOVERY SCHEMES

In this section, we apply techniques for the detection, isolation, recovery, and/or masking of the security threats presented in Section 3.2. Only those faults and errors from Section 3.2 that have viable solutions are included.

---

[4]Classical computer security technology uses the term integrity for the prevention of unauthorized modification of information and programs. This is in contrast to this studies use of integrity to mean design correctness or proper service.

[5]Objects can only be modified by operations in the domain that its type places it in. Subjects must have access rights to domain operations in order to invoke them.

3.4.1 NON-INTEGRITY CAUSED DENIAL-OF-SERVICE ERRORS

This section presents a scheme to deal with the fairness aspect of non-integrity caused denial-of-service errors. Fairness is often used for service sharing policies (and mechanisms) that eventually—no stated time limit—provide a fair allocation of services to users [Fran86, p.4] [Yu88]. However, using the definition of denial-of-service with hard time bounds [Glig83] requires the meaning of fairness used in real-time applications. Thus, from now on, we will use fairness to mean service allocation with the following quantitative measures: (a) amount of service used, and (b) a specific time interval [Jaha86] in which a service can be held by one user.

Denial-of-service cannot be prevented solely by a fairness policy, since it cannot prevent a conspiracy by a group of users from monopolizing a shared resource [Glig85a] [Yu88]. However, the extended notion of fairness presented in this section applies to the following problems:

- A user, holding some service, becomes blocked or terminates before releasing that service. This can result in a service becoming unavailable for an arbitrary period of time [Yu88].
- A user monopolizing the entry queue to a service.
- Service-generated denial-of-service caused by requirement, specification, and algorithm faults that result in undesired interuser dependencies (see Figures 3.9 to 3.13).

Thus, our fairness policy actually contains some aspects of "user agreements" as presented in [Yu87] [Yu88].

One computer security technique addressing denial-of-service in operating systems and networks is based on the construction of models of system behavior and on how services are provided [Cerf85] [Glig85a] [Yu87] [Yu88]. These models are analyzed in order to detect weaknesses in the system design that could lead to denial–of–service (e.g., incompatible resource allocation mechanisms and policies [Glig83]). This approach has the significant benefit that no covert channels are added to the computer system. Run–time mechanisms, such as resource bound enforcement, will likely add channels [Glig85a]. Thus, great care must be used if run–time means are employed to cope with denial–of– service (i.e., limit channel bandwidth, and/or ensure the channels are very noisy).

[Glig85a] actually utilized basic concepts in fault tolerance, that is, the detection of an undesired state (here denial–of–service), and then recovery via the use of an alternate service which provides the save service (i.e., redundancy). A count down timer was set when a service was first acquired, and when it expired denial–of–service was detected. This approach is clearly straight–out of fundamental fault tolerance concepts [Renn84].

The scheme introduced here does not rely on detection, but on prevention of denial–of–service. This approach is used because once denial–of–service is detected the error turns immediately into a failure.[6] Concepts from resource allocation in real–time scheduling [Hami87] are used to guarantee access at a specified time. Action is taken before a service is denied to ensure that a waiting user will be able to use the service in question.

General sharing rules are essentially that of forced fairness:

(1) Services can be held for as long as desired if no other authorized user requests them. Here, denial–of–service can only happen if other users are waiting for the busy service. This is different from the definition appearing in

[6]One of the reasons that this occurs is that some user has witnessed improper service at a service boundary.

[Glig83], which does not require a waiting user. This was chosen because it appears to be more flexible and usable for a real-world computer system.[7]

(2) Once a request is made that cannot immediately be satisfied, because the requested service is being used by another authorized user, a timer is set for the IDS for that service. [Glig83] sets a watchdog timer any time a service is used. IDS stands for imminent denial-of-service: MWT = IDS + time to release the service (TTR). TTR is needed because all services include some fixed time for deallocation (e.g., memory buffers need to be zeroed when released,[8] process context switch).

(3) When a timer expires, enough of the requested service is automatically freed to satisfy the waiting user(s) (i.e., the system deallocates only those services needed from the current users).

(4) If the service is released before the timer expiration, then the timer is turned off.

(5) What if multiple users are waiting for the same service, including entry into the computer system? Authorized users granted access to a service have a guaranteed minimum time of IDS, if desired. Prevention of too many users in a system can be done by setting bounds on users: (a) bound the number of users or network traffic being processed by the system, (b) maximum time that a user can be in a system, and (c) bounds on the number of requests from one source (e.g., number of packets sent from one host in a network). Of course, this includes bounds on the amount of resources (e.g., memory), that any subject can

[7]Any covert channels introduced by this approach will be hard to use due to a great deal of scheduling noise.

[8]When a buffer is deallocated any remaining sensitive data must be removed before allocation to a new subject.

possess (i.e., bounds on a particular resource or group of resources held at any one time [Glig83]). These bounds will lead to the system idling at times. Currently, this seems a necessary evil in order to guarantee access to the system before some MWT expires.

(6) This approach of forcing held services free changes the way services are used in current computer systems. The authorized user, who may lose the acquired service (resource) before his/her work is done, must save its state so that work can continue where left off (i.e., once the service is reacquired). This is identical to the time quantum in CPU sharing. State saving is limited to services with small states for efficiency reasons.

(7) Forced fairness can be improved if the TTR is decreased. This can be done for some services by providing immediately available spares. The used service is still forced free and is used to replace the spares in the spare pool.

(8) Forced fairness imposes an underlying policy on all applications no matter what higher level sharing policies are used.

(9) In order to respond to a real need of guaranteed throughput (e.g., FTP in ARPANET) the concept of deliberate, degradable service sharing is used. First, there are <u>strict</u> bounds on the amount of, total frequency of, and frequency of one requester for degradable sharing. Also, any service must have several active copies in order to allow it to be degraded in the first place.

A shared service (e.g., network path) can degrade (temporarily) to a private resource for a time much longer than MWT. It must <u>eventually</u> (a liveness requirement) return to a shared resource, unless non-deliberate system degradation prevents it. Now, since this service is private it cannot be denied to any other authorized user [Glig83]. Obviously, limits on allowing degradable service sharing are essential in order to prevent denial of a specific level of performance.

(10) Lastly, we could take this notion of forced fairness to the extreme, and end up with a frame-based real-time system with the following properties: (a) a process runs for multiple frames, and (b) the allocation and deallocation times of all services are predefined (off-line generated) and strictly enforced (i.e., services are freed even if no user is requesting their use).

3.4.2 COMPUTER VIRUS ERRORS

Figure 3.1 is a fault tolerance oriented characterization of the behavior of a computer virus. Initially a computer virus can start as a special type of Trojan horse that injects or infects an executable file with a virus. The Trojan horse is a deliberate design fault, and causes an error by changing the state of the executable file resource. Next, the infected executable spreads or propagates the error to other executables. Thus, the error becomes the fault causing other errors, and a typical error propagation occurs just as it does in the case of a random (i.e., accidental) fault. The characterization of a virus as both a fault and an error indicates that viruses should be countered with two mechanisms, rather than just one.

Figure 3.2 provides a somewhat different perspective by indicating the types of damage a computer virus can cause. The figure shows that a computer virus design fault can potentially cause the following errors: loss of integrity of function and data, unauthorized modifications of programs, unauthorized disclosure, denial-of-service, and spoofing. Note, that "DAC" in the figure refers to discretionary access control.

The two life stages of a virus can be detected and recovered from differently. The deliberate design fault via a Trojan horse can be masked out with the use of N versions of a program (e.g., 3 versions of a compiler, see Sections 3.4.6, and 3.6). However, since NVP is too expensive to be applied everywhere, it must be accompanied by a mechanism that can detect the computer virus in its error stage. A scheme to detect and recover from the viral infection is presented in this section and is an extension of Program Flow Monitors (PFM) [Mahm88].

PROGRAM FLOW MONITORS

A PFM is a concurrent error detection mechanism that can be used in the design of a fault-tolerant computer. It is basically a watchdog processor, which is "a small and simple coprocessor used to perform concurrent system-level error detection by monitoring the behavior of the main processor" [Mahm88]. It is used to detect control flow error due to transient (e.g., single event upset) and permanent faults.

Detection of control flow errors is done by comparing dynamic characteristics of program behavior with the expected behavior. One approach is to associate a signature to a sequence of assembly language statements that do not contain any control flow instructions (e.g., branches, subroutine calls). The signatures are derived from the assembly language statements. After generation, the signatures can be stored in a control flow graph (CFG), embedded graph program, or embedded in the executable code. The signatures and control flow graph are generated by a compiler and linker [Mahm88].

As a program runs on a CPU, the fetched instructions go through a signature generator which is based on a linear feedback shift register (LFSR) [Bard87]. Thus, a signature is computed by a given primitive polynomial (e.g., $X^{16}+X^{12}+X^{3}+X+1$). When a control flow change instruction passes through the signature generator, the current signature value is passed to the PFM. The PFM then compares the run-time generated and link-time generated signatures, and a disagreement indicates an error condition. If a control flow graph is used, then it is traversed as these signature comparisons are made.

The applicability of a PFM-based scheme to the detection of computer viruses is based on the observation that actions of a virus also represent an invalid sequence of instructions. However, the basic PFM schemes must be extended to prevent a virus from hiding from it.

EXTENDED PFM TO HANDLE VIRAL INFECTIONS

The present PFM schemes are designed to detect random physical and possibly some design faults, but not deliberate faults. Thus, the existing schemes are susceptible to all but a few viral attack scenarios.

The first weakness against viruses is that PFMs use only one primitive polynomial to compute all signatures. Thus, a computer virus fault compiled on the monitored machine will have valid signatures generated for it. If a CFG is used, then the virus would have to add its signatures to it.

A computer virus error propagating over a network may not have valid signatures, and thus, would be detected by even the existing PFM designs. However, the backward recovery mechanisms used with a PFM (e.g., rollback) would end up mistaking the virus as a permanent fault. This inability to distinguish between viruses and random faults is the second weakness of existing PFM designs. Any PFM-based scheme must have a recovery approach that can identify a viral attack, since the recovery action is different for transients, permanents, and viruses.

The following five extensions are made to a PFM scheme that utilizes a control flow graph (CFG) or embedded graph program:

(1) The signature generator must be able to employ many different primitive polynomials (many such polynomials exist, see [Pete72] and [Bard87, p.71]). This is easily done by constructing an LFSR with sufficient D-flip-flops, XOR gates, and feedback loops to generate an entire range of polynomials. The PFM specifies to the LFSR which polynomial to use by enabling/ disabling XOR gates and feedback loops. The polynomial is represented as a 32 bit wide vector that is latched at the LFSR. The bits of this vector control the enabling /disabling.

(2) The compiler and linker pair must randomly assign a primitive polynomial for each compiled program. This polynomial must be protected from disclosure and modification. Thus, the polynomial bit vector can be stored in the CFG along with the link-time generated signatures, and then the entire CFG is encrypted.

(3) Immediately before program execution the PFM must decrypt the delivered CFG to obtain the pre-calculated signatures and the polynomial. Thus, this approach must also provide management of different encryption keys per CFG, and must ensure executable file - CFG association. Once the polynomial bit vector is obtained, it is transferred to the LFSR. Note that the executable file can itself be both readable and writable.

(4) All I/O operations are atomic. They are performed only if the signature comparisons for their code sequence is valid. This feature blocks the infection capability of the virus. For fault-tolerant computer systems that use backward error recovery this is a necessary requirement, since most I/O operations cannot be rolled back without adversely affecting the service.

(5) At least two features are required in order to apply our modified CFG scheme to general purpose computers: PFM local memory capable of holding

several CFGs [Toma85], and detection of context switches. Instead of detecting context switches the main processor could inform the PFM of a switch (e.g., over private data lines from CPU to PFM). These suggestions would be validated by the PFM before acceptance (e.g., context switches can only occur if the operating system's CFG is being used).

(6) The PFM will store each program code sequence over which it computes a run-time signature. If a mismatch between the run-time and link-time generated signatures occur, then the captured code sequence can be used by diagnostics to facilitate fault location, or used to indicate part of the detected virus code. (Note, a location of the virus code will be pointed to by the current value of the PFM's program counter emulator.) If no mismatch occurs, then this captured code sequence is overwritten by the next sequence. It is expected that these code sequences should range from 5 to 20 words in length [Schu87] [Mahm88].

(7) The current PFM designs concentrate on error detection and do not explicitly address the methods of subsequent recovery. However, details of recovery are important for our application of PFMs, since we need to distinguish between virus errors and physical faults. Upon detection of an error condition the program's execution is resumed with a rollback, and proceeds from a rollback point in the program that immediately follows a previous signature check.

Cohen [Cohe85] has shown that precise detection of a virus by its appearance or behavior is undecidable. Thus, we rely upon imprecise behavioral detection, which can result in false alarms or some undetected viral infections. We choose a recovery technique that will provide a high probability of correct virus identification, with a few false alarms rather than undetected cases.

The rollback procedure helps in the identification of the type of fault as follows. If the initial error was caused by a transient fault, the recomputation will succeed, and the program will continue on after a successful signature check. However, if the initial error was due to a permanent fault or a virus, either will still cause an improper dynamically generated signature after the rollback. These dynamically generated signatures cannot, by themselves, be used to distinguish a permanent fault from a virus error. For example, a virus error could be designed to alter its control flow upon each execution—even after rollback—thus generating a sequence of signatures similar to those caused by many instances of permanent faults (i.e., mimics the error behavior).

To further distinguish between permanent faults and virus errors, diagnostics are run when a rollback immediately results in a second error condition. If diagnostics do not locate a fault, then a high probability exists that a virus error has been detected. This imprecise method may wrongly identify a hard to locate intermittent fault as a virus. Nevertheless, the spreading and planned malicious actions of a virus are prevented. The executable file and its run-time image, both of which can be infected, are clearly identified, thus allowing human analysis. (Note, diagnostics could be PROM based in order to prevent its infection.) Depending on the infection mechanism and damage done, recovery from viral infection can be done by recompilation of a program, reload of a binary executable file from backups, or creation of a new run-time image.

For computer architectures without effective process isolation, the memory address for writing to memory can be monitored by the PFM. This will detect a viral infection of an executable during run-time by noticing that the write address is outside a process's address space (e.g., a block move of virus code into another process's unused stack space). This approach is a design option of a PFM, not a real extension of the technique. In fact, all externally visible actions of a CPU can be monitored by the PFM.

A PFM-based virus detection approach offers some significant advantages. First, it protects an executable even during run-time, while the schemes presented in [Pozz86] and [Cohe87] do not provide this protection. Second, it also provides detection of errors caused by physical, and possibly certain design faults. Third, standard PFM schemes can be extended for virus detection at a modest additional cost. Fourth, no run-time performance degradation occurs, after CFG decryption. Finally, the PFM is virus proof, since all of its components are either hardwired or ROM based, and the PFM local memory, as well as the LFSR can only be accessed by the PFM.[9]

The additional costs of the PFM-based approach are as follows: (a) the compiler and linker pair must assign polynomials; (b) the CFG should be encrypted, and the keys for each CFG must be managed and protected; (c) modifications to existing compilers and linkers are needed; and (d) extra mechanisms for atomic I/O are required.

The scheme presented in [Denn86] also monitors system behavior to detect viruses and other threats; however, it does that at a course-grain level of monitoring. The virus detection approach described above utilizes a fine-grained monitoring of program control flow for detection.

A basic—and simple—architectural solution exists for computer viruses that infect executable files and the run-time image of a process. First, all executable files stored on disk or resident in RAM must be <u>execute-only</u>. Second, only execute-only code segments can be run by a CPU.[10]

---

[9]Note, that extended PFM, by itself, does not protect against denial-of-service due to repeated infection of the same executable.

3.4.3 LOSS OF INTEGRITY OF FUNCTION ERRORS

Most software fault tolerance techniques are applicable (see Chapter 2): NVP [Aviz85a], Software Safety [Leve86], and possibly persistent process-pairs plus transactions [Gray86] (i.e., deliberate software design faults appear to be similar to Heisenbugs in several aspects). Hardware design diversity [Aviz86] can be used to tolerate deliberate hardware design faults.

Insider-generated denial-of-service can be prevented by the use of design diversity in software and hardware. This occurs because loss of integrity of function results directly in denial-of-service (see Figures 3.14 and 3.15a).

3.4.4 LOSS OF INTEGRITY OF DATA ERRORS

Several fault tolerance techniques address the age of data and the integrity of data structures. One technique that prevents the use of stale data is validity timestamps from fault-tolerant real-time systems [Kope85]. A timestamp is associated with each data item, and should only be accessed by the TCB. It contains the time in the future when the data item is no longer meaningful. Once the data item's time expires, its value is automatically removed.

Thus, this technique is effective against a malicious subject using valid but out of date data for updates. For example, a data item can be valid if it is within reasonable limits (e.g., a person's age is greater than 0 but less than 120), but incorrect because it is old data.

Structural integrity of data structures is important. Two techniques that at first appear useful are: (a) periodic data structure audits [Wall84] test that a data structure is intact (e.g., no dangling pointers), and (b) redundancy in data

[10]The Intel 80286 and 80386 already provide such protection to a process's run-time image.

structures [Tayl80] may enable detection of data structure modification and correction of invalid changes. However, these techniques seem too weak to defend against deliberate design faults, which will try to hide their malicious changes to data structures.

3.4.5 UNAUTHORIZED MODIFICATION AND SPOOFING ERRORS

A fault tolerance technique that is applicable to unauthorized modification is to maintain multiple copies of information and programs, such a technique is fragmentation-scattering [Frag85] [Desw86]. This makes it harder to change all the copies for two reasons: (a) they may be hard to find, and (b) it is easier to detect multiple unauthorized modifications via security auditing techniques.

Current computer security techniques to protect against this threat are: integrity-lock [Grau84], discretionary access controls (e.g., privileged modes [instruction subsets] and rings which limit data access, and capability based addressing [Denn82]), integrity controls such as MLI (i.e., multilevel integrity, based on Biba's integrity levels [Biba77]), and LOCK's type enforcement.

Spoofing errors are similar, in that, they may be masked out by use of multiple (redundant), independent command sources with voting. Current computer security techniques include: authentication, trusted path [DoD85a], trusted channel [DoD87], encryption, and access controls.

3.4.6 TROJAN HORSE, COMPUTER VIRUS, AND TRAP DOOR FAULTS

Deliberate use of a storage and/or timing covert channel (e.g., by a Trojan horse) results in unauthorized disclosure of sensitive information. Software fault tolerance, specifically NVP, can detect and prevent the use of some covert channels (see Figure 3.4). This was also observed in [Dobs86]. For this scheme to work the action that uses the covert channel must be voted on (see Chapter 5). However, NVP does not seem effective against many forms of covert timing channels (e.g., the amount of CPU time spent once a process is scheduled can indicate a binary 0 or 1 [Scha77]). Current computer security techniques include: minimizing a covert channel's bandwidth, locating and removing all unnecessary channels, formally proving that they are not used, and audit all channels [DoD85a].

Using NVP is also applicable to all actions taken by a Trojan horse. Current computer security techniques include: MLS/MLI design, strengthening discretionary access controls [Smit86] [Lai88], integrity-lock [Grau84], LOCK type enforcement, and auditing.

The repeated infection of the same executable file (or running process) by a computer virus fault can lead to a special type of denial-of-service. Recovery involves a program reload from some uninfected backup. Denial-of-service is possible now in two cases: (1) program reload exceeds the MWT, or (2) the virus corrupts the reloaded program fast enough in order to prevent any significant program progress (i.e., the program results are not available before some MWT). Note, that only when the computer virus is treated as a design fault is this denial-of-service error prevented (i.e., assuming executables are writable, of course).

Deliberate implantation of a trap door, by a trusted engineer, into an operating system or hardware component of a computer system can be countered by design diversity with consensus voting. For example, N-diverse versions of DEDIX [Aviz85b] can tolerate trap doors placed in less than a majority of the DEDIX nodes (i.e., if DEDIX actions are voted on).

3.4.7 SPECIFICATION AND ALGORITHM FAULTS

Typical techniques to handle specification faults in fault tolerance are to use N versions of a specification [Rama81], and to deliberately include redundancy in the specification. Current computer security techniques rely on fault avoidance and are based on automatically producing a prototype from a specification, and symbolic execution of a specification [Kemm85].

It is important to properly represent maximum waiting times (i.e., MWT) for denial-of-service requirements in a formal specification language. In [Wing86], Ina Jo is enhanced with temporal logic in order to be able to prove time, precedence, and safety properties of computer systems. The following is from [Wing86]: "An example of liveness requirement for a computer network system is that no message should be indefinitely delayed at a node before being serviced or forwarded." Liveness refers to ensuring that something good eventually happens, and is represented by the temporal operator eventually.

The above example from [Wing86] comes close to stating a denial-of-service requirement. However, representing denial-of-service requirements with liveness is inadequate, since denial-of-service occurs only after a specific time period has elapsed [Glig85a]. The following example should make this clear. Using the specification example above from [Wing86], a network message that happened to be a warning that the Russians where invading the U.S. could, for example, arrive over six months late and it would still satisfy its specification. However, it is clear that there is a specific time period in which that message must be delivered to its destination. Beyond this time period denial of message delivery occurs.

A better approach for specifying denial-of-service concerns is the use of Real-Time Logic (RTL) [Barb86] [Jaha86] [Jaha87]. Here explicit time variables (absolute and relative times) are included in a specification which can exactly indicate a time limit for a function (e.g., a sample specification: $\forall i$ [$t1 <= @(\downarrow A,i) - @(\uparrow A,i) <= t2$], where t1, t2 are time variables (an integer value), "$\uparrow$A," "$\downarrow$A" represent events, "$\uparrow$" indicates the start of an event, "$\downarrow$" indicates the completion of an event, "i" indicates the ith occurrence of an event, and "@(e,i)" is used to represent the real time of the occurrence of event e_i. Therefore, this sample specification simply restricts the execution time for operation A to start at or after t1 and end at or before t2.).

Algorithm faults can be countered by fault tolerance techniques such as: robust algorithms, and functionally rich algorithms [Bast85] [Abbo87]. Functionally rich algorithms are designed with functions that have overlapping capabilities (i.e., if one function is lost another can achieve the same result via a different approach). No current computer security techniques other than fault avoidance exist.

3.5 FAULT-TOLERANT, SECURE COMPUTER ARCHITECTURES

In this section we will address the issue of how to make the security features of a computer system fault-tolerant (i.e., fault-tolerant security [Turn86]). In Chapter 4 we will explore the security issues resulting in adding fault tolerance features to a computer system (i.e., secure fault tolerance [Turn86]). This separation is made in order to keep the complexity of the issues manageable and to keep concepts clear.

Current security certification techniques [DoD85a] do not adequately address physical faults, and as such, are not considered in the evaluation process. However, we are concerned with security in the presence of accidental physical faults, and so we extend the requirements in [DoD85a] beyond the current criteria (i.e., beyond A1). Additionally, the current security certification techniques do not adequately address design faults (in software and hardware), since they rely solely on formal methods (e.g., verification, specification, and models [Cheh83] [Kemm85]). This section concentrates on the effects of accidental faults on secure computer systems, since deliberate faults are discussed elsewhere.

The PRIME project [Fabr73], is an early example of a computer designed to maintain process information separation for data privacy in the presence of faults. PRIME was a multiprocessor, multimodule main memory computer. Single hardware or software faults were to be detected by: (1) all actions required approval of two executives—the local executive on an application processor and the one control executive on the one control processor, (2) application processes can have monitoring processes on separate processor hardware, which verified interprocess communication (IPC) messages, and (3) redundant representation of main memory and disk access mappings—one at the executive level and one at the hardware component level (i.e, each main memory page, and disk cylinder contained a process-id like marker).

The PRIME design was to detect and recover from illegal access of virtual memory, disk space, mixing of IPC message contents, ensure source, destination pairs on IPC, and to ensure proper object reuse (i.e., clear a data item before given to a new user). It assumed nonmalicious hardware and software system components protected from outside attack.

A later example, of protecting the access control data portion of a secure computer's critical state, from both physical and some design faults, is reported in [Namj82]. A watchdog processor, containing a copy of a running program's capability lists, monitors all accesses to main memory made by a central processing unit (CPU). Accesses by the CPU, that do not correspond to the capability lists stored in the watchdog processor, trigger a recovery sequence. This is actually, just another example of a run-time monitor as presented in Section 2.3.2.

Contradictory requirements between fault tolerance and security, as discussed in Section 3.1, lead to the following design approaches: (a) the use of fault masking to prevent a security relevant error [Turn86], that would violate the system's security policy, and result in a failing system,[11] (b) the use of nearly (i.e., to a defined coverage(s)) fail-stop or fail-secure [Molh73] computer subcomponents,[12] thus also preventing an error, and (c) the redefinition of a security policy (e.g., mandatory access control) to allow a security relevant error to occur.

A few comments are needed before we proceed. First, option "b" is very hard, since current fault tolerance technology cannot provide fail-stop capability past a limited probability. Second, option "c" is also hard, since detecting security relevant errors (e.g., top secret data in a secret file) may not be possible. For example, a top secret file has a security label on it, but if part of its contents are copied into a secret file, then that data will no longer have an associated label, making it impossible to detect its presence. Plus, this option is likely to be either rejected by the security community as unsafe, or take many years to be appropriately defined. Lastly, fault-tolerant systems provide

[11]The meaning of proper service is application dependent, and for many security policies an error directly leads to a system failure (i.e., a security violation) [Bell75] [DoD85a].

[12]Due to Jacob Abraham, University of Illinois.

proper service with two caveats: (1) they only handle fault classes chosen in the design phase (this is a reasonable limitation), and (2) proper service is always with respect to a probability. Secure systems work more on absolutes (e.g., the TCB must be tamperproof [DoD85a]). An exception is in providing assurances of a proper design, since requirements cannot ask for beyond state-of-the-art tasks.

OPTION A — FAULT MASKING: Use forward error recovery only, for example, [Dobs86] describes releasing misinformation after a security violation as a compensating action. In [Turn86], as already mentioned, a better approach is to restrict fault tolerance techniques totally to masking. Voters can be cascaded (i.e., each computer subcomponent masks faults), and if a voter is overwhelmed the resulting errors should be masked by other voters further down in the hierarchy. Such generated errors must not be security relevant[13] (i.e., against the system's security policy). For example, if unauthorized access can occur by the following error propagation: $<e_1, e_2, ..., e_n>$, "e_n" being the disclosure, then a system failure could be prevented by stopping error propagation before e_n (i.e., only e_n is security relevant). Of course, the system must also be able to recover from the damage done.

Hybrid redundancy (i.e., NMR with standby spares), can degrade to two active elements. One additional failing element will force an NMR computer subcomponent, and possibly a system, into a fail-safe state. Another approach can be Pair and Spare, as used in Stratus machines [Frei82]. Essentially, all that is required for this fault-tolerant, secure design, is to blindly place a TCB

[13]Security relevant errors may be the result of non-security relevant errors (e.g., loss of integrity of function, which is not directly a security relevant error, but can lead to unauthorized disclosure of information).

on top of such a hardware base.

The last voter in a hierarchy should be placed at the same boundary as the information flow boundary defined by the security mechanisms[14] (e.g., boundaries between sensitivity levels enforced by a TCB, or the domain and type checking in LOCK). Any error produced at this boundary would be equivalent to the e_n in the previous example.

It is important to notice that identifying hardware in which a fault could generate a security relevant error should include all basic hardware support. Software implemented security mechanisms depend on proper service of even the simplest hardware components (e.g., arithmetic operations for comparisons). Currently, most computer security systems implement a significant portion of security mechanisms in software.

This simple, brute force approach can be used for all types of secure computer systems (i.e., C1 through A1), since the fault tolerance approach is consistent with, and helps achieve, the current state-of-the-art security policies and models (e.g., [Bell75]). Interestingly, this fault tolerance approach is behavioral [Shen86] [Gill87], since it takes the application into account. However, the resulting solution, at first glance, appears structural.[15] This is, obviously, due to the fact that the application is itself very ridged in its defini-

[14]Observation of matching security boundaries to fault tolerance error containment boundaries is due to Jacob Abraham, University of Illinois.

[15]Structural Fault Tolerance—Applying basic fault tolerance techniques to preserve a computer's processing capabilities and critical state, while ignoring the behavior of the application (e.g., an occasional error in a signal processor is not a problem, since over time it is smoothed out via finite element mathematics).

tion of proper service.

It is important to note that errors, e_1, ..., e_{n-1}, need not be handled by masking techniques (i.e., detection and recovery can be used). Control flow errors caused by transient faults can be a good example of this. The control flow error can be detected and recovered from, via a program flow monitor, since by itself, it does not violate a security policy. A possible, e_n, resulting from a control flow error is unauthorized disclosure of sensitive information (e.g., when a control flow error forces a jump past an access control test). This, e_n, will be prevented by quick detection of the control flow error.

OPTION B — FAIL-STOP: Computer subcomponents would be built using self-checking logic [Cart68] [Renn84], and software aimed at detecting hardware problems [Hua86]. Fault-tolerant systems do not rely solely on this approach, because it makes the false assumption that errors will not propagate past the self-checked components. With this approach error recovery is significantly simplified (e.g., a node in AOSP would halt due to a fault, rather than babbling on an internode bus). If only a limited coverage is required, then this approach may be feasible, and it is certainly less expensive than fault masking.

OPTION C — ALLOW SECURITY RELEVANT ERRORS: This approach would allow the use of backward error recovery techniques. One such approach, that seems directly applicable, are atomic actions. If a security relevant error propagates, then the chance of detecting it becomes rather small. Therefore, it must be detected very close to where and when it first appeared. That is, a small detection latency is necessary, or in other words, small error containment boundaries are needed. This will facilitate a complete recovery from the error (i.e., remove the insecure state and repair the cause of the error). Obviously, sufficient hardware is needed to achieve this concurrent error detection.

For this approach to work, all security relevant errors must be recoverable. A non-recoverable security event is when an outside user (i.e., a human) is able to see the effects of an error. The difficulty of this is not as obvious as it first appears. What if, one of the actions of a security relevant error propagation, involved the use of a covert channel to distribute leaked data? The use of the covert channel must then be recoverable. This involves restoring the state of the receiving process and the resources used as the channel.

3.6 DESIGN OF A SECURE[16] DEVELOPMENT ENVIRONMENT

The design fault stage of a computer virus can exist in a program handling tool (e.g., a compiler), as well as, in an ordinary application program [Cohe84] [Pozz86]. Thompson [Thom84] gives an example of how a Trojan horse in a C-language compiler can implant a trap door into a UNIX login program. In this example, the Trojan horse is particularly insidious in that it is designed to: (a) detect its own compilation (i.e., the C compiler's source code) and then to implant a copy of itself into the generated executable, and (b) the actual code for the Trojan horse can be removed from the compiler's source after its first compilation because of item "a." Part "b" makes the detection of this Trojan horse attack by source code inspection and verification impossible. Currently, addressing the correctness of program handling tools is beyond the requirements for class A1 secure systems [DoD85], and thus, is a hole in any current defense. In summary, a virus can infect a program during an editing session, compilation,

[16]We prefer not to use trusted development environment, since we do not plan to <u>trust</u> any logic to follow a security policy without some run-time assurances.

assembly, linking, or loading.

In addition, assurance of the correctness of hardware and firmware (i.e., the absence of random and deliberate hardware design faults) for the TCB of a system is also beyond A1 for the hardware of both development environments and operational systems. Current research in secure execution environments is directed towards the use of advanced formal verification techniques for both hardware and software [Bevi87]. While this research looks promising, it has the following limitations: (a) it is hampered by the same problems that all formal verification suffer—it does not scale up well to large, complex systems,[17] (b) using the verification language Gypsy [Good86] as a software implementation language would be cumbersome (i.e., Gypsy does not include: global variables, own [static] variables, pointers, or floating point arithmetic), and (c) the approach used for hardware verification does not include timing, nor system behavior in the presence of faults (i.e., the system may be broken into due to deliberately induced faults—not fail-secure [Molh73]).

In this section we propose a design for secure development tools based on N-Version Programming, and secure development hardware based on hardware design diversity. A secure 3-version[18] C-language compiler, for example, would operate as follows. Consensus voting between versions would periodically occur on several items: (a) parts of the local state, (b) temporary output at each

[17]The Gypsy environment [Good84] allows specifications to be verified by run-time validation (i.e., a specification is evaluated to true or false at each program state), as well as formally verified. This can be used to decrease the difficulty of verifying some specifications.

[18]Two version systems are susceptible to denial-of-service attacks [Jose87].

phase of compilation, (c) actions which manipulate files, and (d) the final generated code. The voting locations ("cc-points") and the values to be voted on ("cc-vectors") need to be clearly defined in the compiler's design specification before it was built. (Note, for this to work strict guidelines on code generation and optimization must be provided.) Item "c" above includes the "action voting" extension to NVP, which was introduced in Section 2.3.1 and is further developed in Section 5.1.1.

It has been discussed in [Dobs86] and Section 3.4, that NVP could be used to prevent a program from making use of many instances of covert channels. This fits in well with our use of NVP-based program handling tools, because a Trojan horse in an editor attempting to leak sensitive information would be stopped.

Experiments at UCLA have already demonstrated the feasibility of constructing a reliable NVP text processor [Chen78]. However, an NVP-based tool is just as vulnerable to a computer virus error as any other executable file. Thus, if a virus infects a majority of the versions the NVP scheme would be defeated. So to prevent this from occurring, each version must be monitored by a PFM (see Section 3.4.2), or protected by a scheme such as presented in [Pozz86], [Cohe87].

In summary, the reason why NVP-based program handling tools can counter the design fault stage of a virus, and also many actions of general Trojan horses, is that all maliciously generated actions are masked out by the N version consensus operation. See Chapter 5 for a discussion on the resistiveness of NVP to implanted malicious logic.

To provide secure[16] development hardware traditional approaches to verifying correct hardware function [Glig85b], such as testing and simulation, are also a far cry from providing the needed assurances. In [Aviz86], hardware design diversity is proposed as a technique to tolerate accidental design faults, and should also be applicable to deliberate hardware design faults. One drawback of this approach, is that proper synchronization of diverse hardware components is not trivial.

Lastly, a development environment should contain standard computer security techniques used in systems that allow sharing, these include: configuration control, access controls, and auditing. The final result of the design presented here is a developmental computer system, which is tolerant of both deliberate and accidental design faults. Certainly, the application of design diversity can be expensive, and so in that respect not glaringly better than formal verification. However it has two important pluses: (a) it is a run-time mechanism, and so it is always on guard, and (b) it can be effectively used for an <u>entire</u> computer system today, unlike formal verification.

3.7 HIERARCHIES AND FAULT-TOLERANT, SECURE SYSTEMS

This section should be read using the less strict interpretation of the application of security doctrine (i.e., that an error in a secure system does not immediately lead to a failure, as discussed in Section 3.1).

Presented in [Neum86] is a design approach for critical systems, where criticality encompasses the following system properties: human safety, fault tolerance, high availability, security, privacy, integrity, and timely responsiveness for the entire life-cycle of a computing system. Neumann proposes that a hierarchical layering of a system with allocations of safety, security, and fault tolerance mechanisms at each layer (depending on the function of that layer), will provide a much better system design than is achievable with current methods.

Also presented is a careful analysis of critical functions and how these are placed into the hierarchy. Essentially, it proposes a hard core approach where each layer handles its own critical requirements, and some of the requirements of the layers above it.

One disturbing notion used throughout [Neum86] is the computer security notion of trust. For example, observe its use in the following sentence from [Neum86]: "In any event, design decompositions should be sought that require only a small portion of the system to be trusted (in the broader sense of the critical requirements)." This notion of trusting a piece of software or hardware to correctly perform its function is foreign for fault tolerance practitioners.

On page 916 and 917 of [Neum86], a strawman design for a critical system is based on a TCB that enforces a multilevel secure (MLS) and multilevel integrity (MLI) environment. Limiting this discussion to only the computer security and fault tolerance concerns reveals several problems with this scheme.

If the fault tolerance mechanisms in a particular layer are defeated, either by being overwhelmed by faults or by insufficient coverage, then either of two actions can occur. An error(s) can be detected and recovered from by a layer below the failing one (e.g., voters in DEDIX), or by a layer above (e.g., GOS in AOSP [RADC85]). If the latter occurs then a serious security violation can occur. It is a basic tenet of secure systems, that the level of trust increases towards the innermost layers of a system (e.g., the kernel is the most trusted layer), and that a more trusted layer cannot be compromised from above (i.e., a more trusted layer cannot rely on a less trusted layer for anything). As an example of this, a less trusted layer doing detection and recovery of a failed, more trusted lower layer, can declare false alarms, ignore detection of real errors and thus perform recovery at the wrong times or not at all!

Without using the notion of trust presented above for the MLS/MLI based design, no layer, above a failed layer can detect or recover for it. Additionally, it is unreasonable to assume that any layer will not fail completely, and thus, need help from above. For example, lets put the above discussion in the context of a system based on DEDIX. DEDIX provides most of the error detection and recovery mechanisms for the N versions of application running on top of it. However, such an application can still fail and if it does all DEDIX could do is to shut it down. Still more global detection and recovery mechanisms would be needed if such an N-version application ran in one of the hierarchy layers in Neumenn's MLS/MLI based design [Neum86].

Now, if we allow the use of the notion of trust, then at each interface between layers the detection and recovery mechanisms would have to be trusted. This would result in a need for too much trust to be practically verified. Also, detection and recovery mechanisms often deal with very strange combinations of faults and errors. Exact details of each combination expected and how it is handled will not be easy to define, if possible at all. This detail will likely be needed in order to have trust in its behavior. (This is essentially detailed verification and validation of detection and recovery mechanisms.)

An interesting observation to make is that the hierarchical design of a secure system is meant to keep intruders out, while the hierarchical design of a fault-tolerant system is meant to keep errors in (i.e., from propagating). A question to ask is: Can we place all the fault detection and recovery mechanisms into the innermost trusted layers of a system? One problem with this, is that in such a hierarchy lower layers are not supposed to know what the higher layer application is (also see Chapter 4). Lower layers only provide a virtual machine interface for the higher layers to use. Lastly, none of the important issues brought up in [Turn86] are addressed in [Neum86].

3.8 GRACEFUL DEGRADATION OF COMPUTER SECURITY

Fault-tolerant computer systems gracefully degrade in performance (i.e., go to a lower level of service) when all spare units for one resource are depleted, and less than the minimum required units of that resource are working. Additionally, the error detection and recovery coverages can degrade at the same time as performance, since the hardware used by the fault tolerance mechanisms can be lost with dwindling hardware and software capabilities.[19]

The idea to apply this general concept to computer security was first presented in [Turn86]. Its purpose is to provide high availability to a system, with some degree of computer security, and thus partly avoiding undesired exposure to attack. Security can be forced to degrade due to some hardware and/or software fault, causing the loss of a required security mechanism in that system's initial security evaluation class (i.e., division: C,B,A, classes: C1, C2, B1, B2, B3, A1 [DoD85a]). As an example, loss of a trusted path capability due to a physical, permanent failure of a hardware component, could force class A1, B3, or B2 system to degrade to a class B1 system. In this example, once the failure has been required, the secure system could then perform a recovery to its original evaluation class. However, in the future, it is likely that not all secure computer systems will be accessible for maintenance (e.g., the SDI).

The DoD security criteria[20] [DoD85a], uses four basic types of system

[19]An exception to this, is when self-checking computer modules are employed. Upon degradation, error detection and recovery are preserved, since each self-checking module contains all its <u>local</u> detection and recovery mechanisms [Renn78].

attributes in an evaluation: security policy enforced (i.e., the security requirements), accountability of system subjects, assurances of proper design and implementation, and documentation. A fault in a deployed secure computer system, specifically in the implementation of the security policy, accountability mechanisms, and/or the system architecture section of assurances, can force security degradation.

Degradation of computer security occurs from a system's initial evaluation class, downward in the defined DoD security criteria's divisions and classes per division.[21] In a deployed, operational system, this will involve a loss of system functionality (e.g., from handling multiple levels of sensitive information, to degrading to a single level of information), flexibility of use, and penetration proofness of the system. In exploring this topic, we will first outline in Section 3.8.2, all possible security degradations that can occur by removing a specific security mechanism required for each class in [DoD85a] (e.g., remove all audit capabilities). Then in Section 3.8.4, we will describe which degradations, out of all enumerated, make sense and can be implemented.

Two fundamental goals must be achieved to allow the use of security degradation. First, security degradation must not allow any compromise. If for example, all security labels in a secure computer where lost due to some unchecked error propagation, then a valid security degradation would not be to logically upgrade all subjects and objects to the highest security level of the previously working system. (It could be done simply by ignoring all mandatory

[20]This determines what type of trust can be placed in the system and what threat environments it can be used in.

[21]This is the approach described in [Turn86].

access controls and relying only on existing discretionary controls.) This is undesirable, because it allows previously unauthorized, less privileged users access to privileged data.

Second, a reasonable recovery time is likely to be required. This requires security degradation to be handled by the computer system (i.e., no user intervention). Later, once the system is operating again in a secure mode,[22] authorized human intervention can be allowed to ensure a tolerable working environment.

An important question to ask is: does a fault or flaw in any of the remaining assurances (i.e., off-line techniques, not including the system architecture section, such as configuration control [DoD85a]) and documentation requirements also force security degradation? In fact, this is feasible, if the inadequacy of the off-line techniques, or of the way they are applied, result in undetected flaws in the security mechanisms.

However, these type of security degradations are very open ended, since a wide class of faults can result from them. These type of faults overlap significantly with the previous work done in this Chapter and Chapter 2, and as such, have already been discussed.

In order to use some of the advanced concepts presented in the previous sections of this dissertation, and also to be forward looking, we include the security evaluation class of A2, which is not yet officially defined and is viewed to be beyond the state-of-the-art. Before we proceed with discussing security degradation, we first define a portion of the run-time requirements that we believe should be included in this security class.

[22]Thus a true trusted recovery, rather than the current concept in [DoD85a].

3.8.1 PARTIAL REQUIREMENTS FOR A CLASS A2 SECURE COMPUTER SYSTEM

First, add to Section 4.1.1 Security Policy in [DoD85a] the following sub-sections: "4.1.1.5 Modification Access Control - The TCB shall enforce a formally specified data integrity policy that ensures either condition 1 alone or both condition 1 and 2. (1) Prevent the unauthorized direct or indirect modification of information and programs. (2) Manipulations on objects must be constrained to well defined and controlled software and hardware. Sequences of object manipulations must be constrained to well defined and controlled sequences.

If only condition 1 is to be supported, then this can be provided by using Biba's simple integrity property and integrity *-property [Biba77] [GC84].

If condition 1 and 2 are to be supported then they can be provided by using a Clark-Wilson based integrity policy [Clar87]. [see Section 2.2.2] The type enforcement mechanism introduced in LOCK [Boeb85a], provides the basic capabilities for implementing this policy."[23]

"4.1.1.6 Integrity of Function and Data - Run-time mechanisms shall be dispersed throughout a computer system's hardware and software to ensure its proper service in the presence of deliberate faults [Jose87]. This includes the integrity (correctness or proper service) of a development environment."

Second, add to Section 4.1.3.1.1 System Architecture in [DoD85a] the following statement: "Errors due to physical faults (both transient and intermittent) in a computer system's hardware, and design faults in hardware and

[23]The Clark-Wilson integrity policy includes condition 1 to protect the well defined, constrained object manipulators (e.g., programs), and the database of authorized manipulation sequences from unauthorized modification.

software shall be tolerated, and thus, maintain a computer system's security in the presence of faults from defined fault classes."

All other requirements, such as, improving off-line techniques (e.g., verification techniques applied to covert timing channels, and enhanced security testing) are not of interest here, since they are purely fault avoidance techniques.

3.8.2 ALL POSSIBLE SECURITY DEGRADATIONS

Computer security can be forced to degrade for the following reasons: loss of some portion of the critical state (see Section 3.3), design fault (accidental or deliberate) in the hardware and/or software of a security mechanism, and computer system degradation due to physical faults. All security mechanisms and policies appearing below where taken from each secure computer class appearing in [DoD85a].

1. Degrading from A2→A1 can occur if any of the added run-time requirements presented above are completely or partially lost.

2. Degrading from A1→B2 or B3→B2 can occur due to any one of the following: (a) loss of the discretionary access control to list individual or groups of individuals who are not to be given any access to an object (enforcement of security policy), (b) loss of the capability of the TCB initiating a trusted path, and that a trusted path is always used when a positive TCB-to-user connection is required (e.g., login, change subject security level) (enforcement of accountability), and (c) loss of the capability of the TCB monitoring occurrences of auditable events, which may indicate an imminent violation of the security policy; if thresholds are exceeded the TCB reports to the security officer, and after which if these events continue take action to prevent such further events

(enforcement of accountability).

3. Degrading from B2→B1 can occur due to any one of the following: (a) loss of a significant amount of labels on system resources (i.e., can be on hardware: ROM, I/O device) that are directly or indirectly accessible outside the TCB, loss of the ability to inform a subject when his/her security level is changed, and loss of the ability of a physical device to have a range of security levels of objects that can be written to it, (enforcement of security policy), (b) loss of mandatory access control over all resources, to only control over subjects and storage objects (e.g., ROM is not a storage object but at B2 it can have a label) (enforcement of security policy), (c) loss of the capability of identifying audit events that may be used in the exploitation of covert storage channels (enforcement of accountability), and (d) loss of the capability to have a trusted path (enforcement of accountability).

4. Degrading from B1→C2 can occur due to any one of the following: (a) loss of a significant amount of all security labels (enforcement of security policy),[24] and (b) loss of process isolation maintained by the TCB via distinct address spaces under its control (operational assurances).

5. Degrading from C2→C1 can occur due to any one of the following: (a) loss of the capability of discretionary access control defining groups <u>of individuals</u>, with controls to limit propagation of access rights, and the ability to include or exclude access at the granularity of a single user (enforcement of security policy), (b) if not defined by a user, the discretionary access control

[24]Loss of object importation from or exportation to a multilevel device (e.g., I/O device, communications network) can be recovered from by using several single level devices. That is, the loss of multilevel devices, by themselves, do not seem to warrant a security degradation.

mechanisms will by default protect objects form unauthorized access (i.e., set initial default access rights so only owner can touch data), (c) loss of all object reuse provisions (i.e., ensure all old data is removed) (enforcement of security policy), and (d) complete loss of audit (enforcement of accountability).

6. Degrading from C1→D can occur due to any one of the following: (a) complete loss of discretionary access controls (enforcement of security policy), (b) complete loss of all capabilities of user identification and authentication (enforcement of accountability), and (c) the TCB is unable to protect itself from modification (e.g., running the Intel 286 in non–protect mode).

Above we have detailed all the security mechanisms that can be lost at run–time, and what the resulting degradations would be. However, this does not, by itself, specify how such a degradation would be done. For example, if given a secure system in division A or B, and a significant amount of all security labels where lost, how would the security degradation to a C2 system be done?

An interesting case is if a class A2 system loses a significant amount of its integrity labels, while the normal security labels are intact. In this case, the security simply degrades to a class A1 system. If the reverse happens (i.e., integrity labels are intact and the security labels are lost), then security degrades to a class C2 system. However, integrity labels can still be used to enforce a type of access control. These type of issues are discussed in Section 3.8.4.

3.8.3 TRANSITIVE CLOSURE OF THE SECURITY DEGRADATION FUNCTION

The point to be made here, is that a sequential degradation down the security evaluation criteria (i.e., A2→A1→B3→B2...) is not essential. Several divisions or classes can be skipped directly depending on the security mechanism(s) lost. The security degradation function (SD) can be represented as – SD: I→P, where I is the initial security evaluation class of secure systems, I = {A2, A1, B3, B2, C2, C1 }, and P are all the possible security evaluation classes to degrade to, P = {A1, B3, B2, B1, C2, C1, D}.

SD is defined as follows: SD(A2) = A1, SD(A1) = B2,[25] SD(B3) = B2, SD(B2) = B1, SD(B1) = C2, SD(C2) = C1, SD(C1) = D. The transitive closure of SD, SD^+ is:

{ (A2,A1), (A2,B2), (A2,B1), (A2,C2), (A2,C1), (A2,D),
(A1,B2), (A1,B1), (A1,C2), (A1,C1), (A1,D),
(B3,B2), (B3,B1), (B3,C2), (B3,C1), (B3,D),
(B2,B1), (B2,C2), (B2,C1), (B2,D),
(B1,C2), (B1,C1), (B1,D),
(C2,C1), (C2,D),
(C1,D) }

This closure shows all possible security degradations.

An example of a security degradation is the loss of a hardware device that ensures that a terminal user can directly contact the TCB (e.g., a special function key on a keyboard), that is, the loss of the trusted path. This can be explicitly represented by another function, SDE (for SD Extended): IxS→P, where S = {set of all required security mechanisms}. Examples of elements in S are trusted path, audit storage covert channels, and mandatory access control. For this example, SDE is defined as: SDE(A2, trusted path) = B1, SDE(A1, trusted

[25]Degradation from A1→B3 can only occur due to loss in effectiveness of an off-line technique. In this section we are only considering degradation due to run-time faults.

path) = B1, SDE(B3, trusted path) = B1, SDE(B2, trusted path) = B1, and SDE(D, trusted path) = undefined. These degradations form the following subset of SD^+, { (A2,B1), (A1,B1), (B3,B1), (B2,B1) }.

As a last example, consider the loss of all audit capabilities. Again for this example, all possible security degradations is a subset of SD^+, { (A2,C1), (A1,C1), (B3,C1), (B2,C1), (B1,C1), (C2,C1) }. Some values of SDE for this example are: SDE(B3, all–audit) = C1, SDE(C2, all–audit) = C1, and SDE(C1, all–audit) = undefined. As done above, SD^+, is used to help enumerate all possible values of SDE.

These two examples, should make it clear that the loss of an essential run–time security mechanism (e.g., audit), forces the security evaluation class to degrade below the security class at which that mechanism was first introduced (e.g., trusted path first introduced at class B2). Degradation should occur even if other higher security class mechanisms are still effective (e.g., loss of adequate discretionary access controls can force security degradation to C1 or D, even if security labels used for mandatory access controls [needed for B1 and above] are intact).

3.8.4 MEANINGFUL AND MEANINGLESS SECURITY DEGRADATIONS

All possible security degradations of Section 3.8.2 are based on the classes (e.g., A1, C2) appearing in [DoD85a]. However, for the following discussion it is more appropriate to discuss secure computer modes of operation, which actually encompass these classes [Air84] [DoD85b] [DoD85c].

These modes of operation include: multilevel security mode which corresponds to, class B1 through A1 systems, controlled security mode which corresponds to, class B1 through B3 systems, system high security mode which corresponds to, class C1 and C2 systems, and dedicated security mode which corresponds to class D and C1 systems. Definitions for each of these modes can be found in the glossary of security terms and concepts.

MULTILEVEL AND CONTROLLED MODE TO SYSTEM HIGH

Degradations that force a secure computer system from a multilevel or controlled mode of operation [Air84] (i.e., basically, this allows multiple levels of sensitive information concurrently in one machine, or network), to a system high mode of operation (i.e., basically, only one level allowed), require the removal of certain subjects or objects in order to avoid compromise. An example of this, is SDE(A2, all-audit) = C1, SDE(A1, all-audit) = C1, SDE(B3, all-audit) = C1, SDE(B2, all-audit) = C1, and SDE(B1, all-audit) = C1.

The loss of an audit facility could be due to, for example, a design fault (accidental or deliberate) in its implementation, or a physical fault in its storage facility (e.g., disk head crash reduces audit archive storage in half). This particular example is justified, since it would be exceedingly difficult, if not impossible, to track down any abuse of privileges by trusted subjects (e.g., use of covert channels), or a general system penetration without an audit facility.

Specifically, to degrade to a system high mode, the highest level of sensitive information to exist in the computer must be picked. Two choices exist: the current highest security level or the lowest security level. If the highest level is maintained after a security degradation, then all subjects below that level must be removed, (see Figure 3.16 (a)), and all security labels in the system are logically deleted. This essential causes all objects to be upgraded, and can easily be implemented by setting all labels to the same value. The

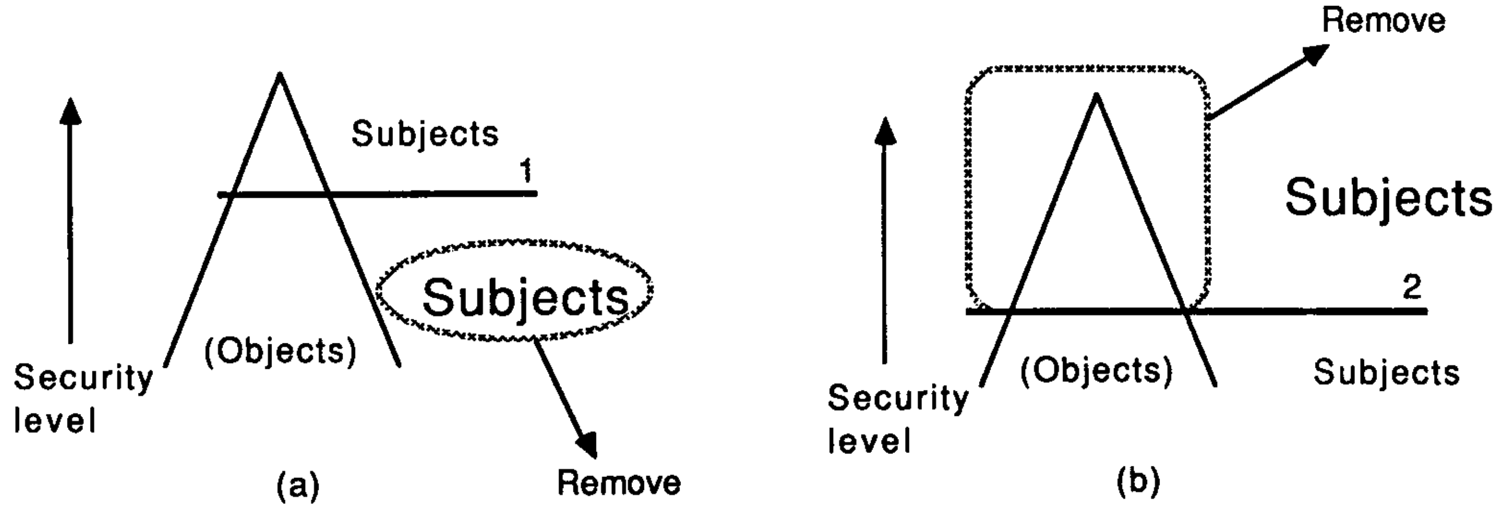


Figure 3.16 Degradation to System High Mode

Multilevel and Controlled mode degradation to System high mode, (a) 1: single level set to highest security level, and (b) 2: single level set to lowest security level. The size of letters and enclosed space does indicate number of items.

degraded system will provide access to all objects for a limited set of users.

If the new highest security level of the degraded system is set to the previously lowest security level (i.e., the later case), then all objects above that lowest level in the system are removed, (see Figure 3.16 (b)), and again all remaining security labels are logically deleted. This causes the entire system to be downgraded in security level, and will then be less useful for the most users.

The following set of security degradations represent all the possible cases for this section and is a subset of SD^+, { (A2,C2), (A2,C1), (A1,C2), (A1,C1), (B3,C2), (B3,C1), (B2,C2), (B2,C1), (B1,C2), (B1,C1) }.[26] Any of these security degradations can occur due to the following loss of security provisions in [DoD85a], and were presented in Section 3.8.2: 5d– loss of all audit, 5c– loss of object reuse, 5b– loss of default access setting, 5a– loss of control on propagation of access rights, and the ability to include or exclude access to a single user, 4b– loss of process isolation enforced by the TCB, and 4a– partial loss of security labels.

This last example (i.e., 4a), actually represents the case of partial loss of critical state information (see Section 3.3). The loss of a significant portion of object security labels can also cause security degradation to a system high mode of operation. This is done the same way as previously presented when the single security level chosen, for the degraded mode, was the current highest level. The loss of a significant portion of subject labels results in the single security level chosen to be the current lowest security level. Thus, if only the

[26]Class D systems are not important, since they represent a total loss of security.

lowest security objects exist in a secure system with subjects of unknown trustworthyness, then no compromise is possible.

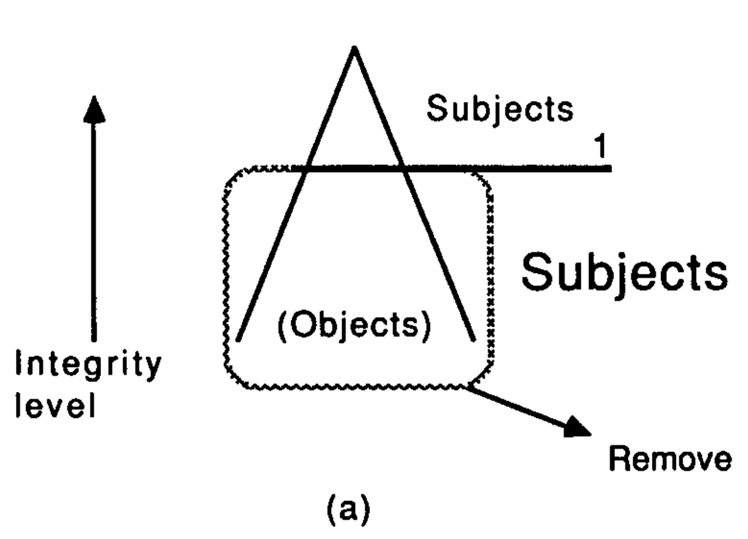
Loss of a portion of the encryption keys in a secure network can result in a MLS network degrading to a SLS network. This loss of encryption keys can occur for a period of time when a key distribution center (i.e., a special node in the network) becomes malfunctional.

MULTIPLE LEVELS OF BIBA'S INTEGRITY TO SINGLE LEVEL OF BIBA'S INTEGRITY

All references to integrity levels in this section refer only to Biba's integrity levels [Biba77]. If the secure computer system before a security degradation was a class A2 system, then degradations might be required to prevent a Biba's integrity compromise or information sabotage [Biba77]. To accomplish this, we need only extend the scheme presented in the previous subsection.

The first possibility is to degrade the multiple level integrity (MLI) system to a single level integrity (SLI) system at a Biba's high integrity level. To do this all objects below the highest integrity level are removed (see Figure 3.17 (a)). This prevents high integrity subjects from being spoofed by low integrity objects. Next, all remaining integrity labels are logically deleted (i.e., set to one value). At first glance this may appear as if the previously low integrity subjects can now sabotage high integrity objects, since write access is possible after degradations. However, since only high integrity objects remain in the degraded system, a low integrity subject will not be able to contaminate a high integrity object with low integrity objects.

The last possibility is to simply downgrade all objects to the lowest



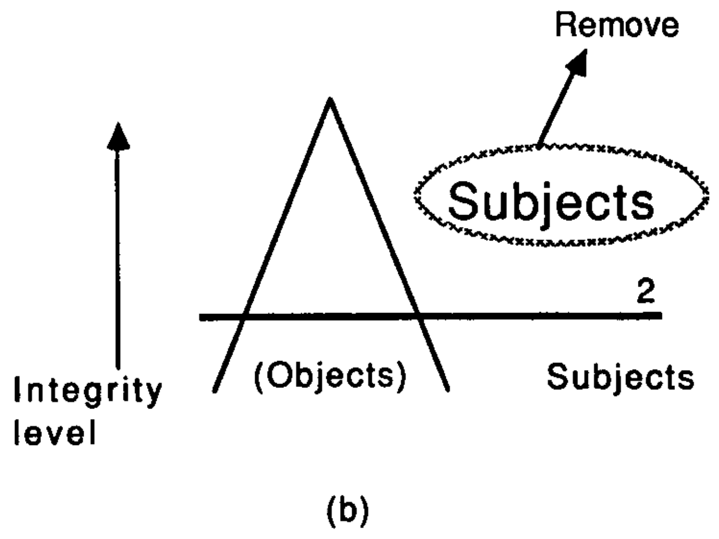


Figure 3.17 Degradation to a Single Level of (Biba's) Integrity

Multiple levels of integrity to single level of integrity degration, (a) 1: single level set to highest level of integrity, and (b) 2: single level set to lowest level of integrity. (All use of integrity here refers to Biba's integrity levels [Biba77]).

integrity level. This requires the removal of all subjects above the new low level (see Figure 3.17 (b)). Now, it does not matter that high integrity objects can be contaminated with low integrity objects, since there are no high integrity subjects to be spoofed. Notice, that these two integrity cases are the dual of the multilevel and controlled mode to system high mode degradation cases, as are integrity concerns the dual of security [Biba77].

The following two possible degradations, { (A2,C2), (A2,C1) }, introduce an interesting example. A MLS/MLI system degrading to an SLS/SLI system. This is possible by integrating the approaches in this section and in the last subsection. However, great care must be taken, if the security levels do not correspond to the integrity levels (i.e., a subject with a high security level also has a high integrity level), then it may be possible to degrade to a system with no subjects or objects. Figure 3.18 a&b describe security degradations were security and integrity levels correspond with each other.

MULTILEVEL MODE TO CONTROLLED MODE

This type of security degradation would be done to reduce the damage of a possible compromise, while still allowing the flexibility of MLS system use. In controlled mode of computer system operation, at most two classification levels of subjects in the system is allowed. However, object classification can range from the lowest security level possible to the system's highest security level (e.g., lowest subject clearance is confidential, then unclassified to top secret objects can exist concurrently) [Air84]. A class B1 secure system can be used for a controlled mode of operation. All possible security degradations to it are a subset of SD^+, { (A2,B1), (A1,B1), (B3,B1), (B2,B1) }.

A plausible example of such a degradation is represented as: SDE(A2,

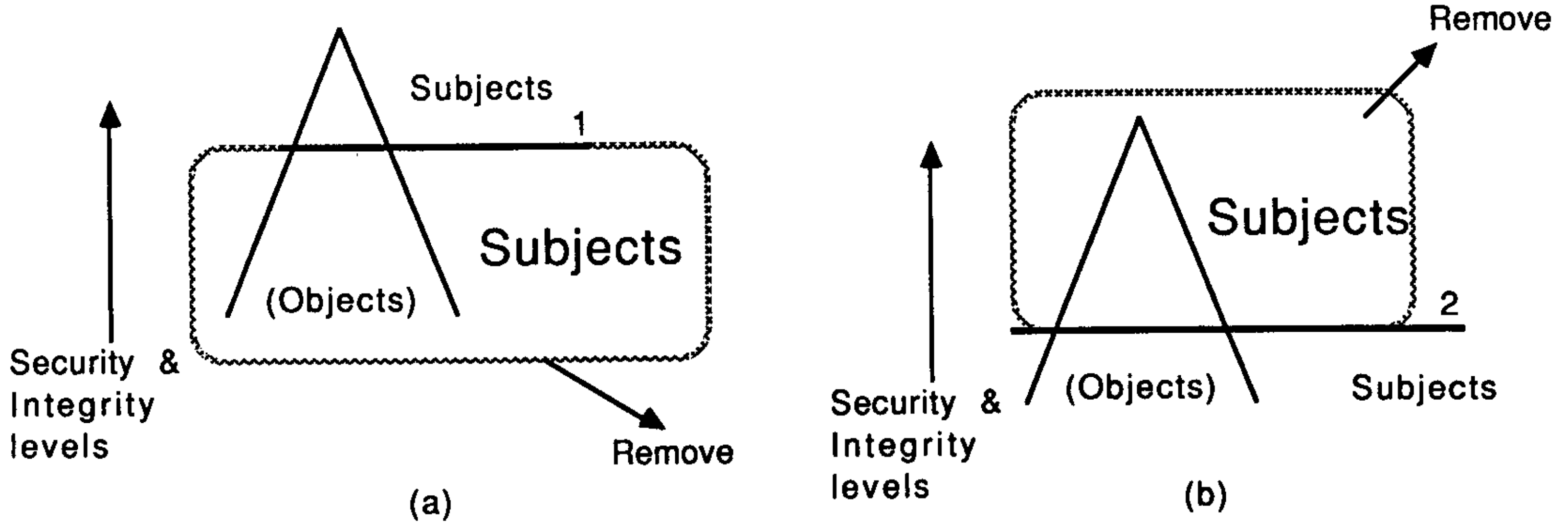


Figure 3.18 Degradation in Security and Integrity

Security degradation with security and integrity compromise prevented: (a) 1: set security and integrity levels to highest level, and (b) 2: set security and integrity levels to lowest level. (All use of integrity here refer to Biba's integrity levels [Biba77]).

storage channel audit) = B1, etc. Allowing, MLS system to operate normally under these conditions can result in a serious compromise. The security degradation to a restricted number of security levels uses the same basic idea of degradation to only one level.

In general, to degrade security to a restricted set of security levels the new lowest and highest security levels allowed must be chosen. All levels in between will also be allowed. First, all subjects below the new lowest security level will be removed. All objects below this level can either be upgraded to the new lowest security level or simply removed. Second, all subjects above the new highest security level are downgraded to the new level once all their state information is reinitialized (i.e., prevents old data from being leaked out—like object reuse). All objects above the new highest security level should also be removed from the secure computer system.

Since controlled mode allows low security level objects, the above upgrading or removing of low level objects can be bypassed. Degrading to a controlled mode does not require logical deletion of security labels, since they are still needed. Instead, they are downgraded by rewriting then with the appropriate new security level.

Specific losses of security provisions that can force these type of degradations, and were presented in Section 3.8.2 are: 3d– partial loss of all trusted paths, 3c– loss of potential storage channel audit, and 3b– partial loss of security labels on system resources (e.g., ROMs, and I/O devices).

It is interesting to observe that the above scheme to restrict security levels can be used for two additional purposes. First, any security degradation between MLS systems (i.e., multilevel mode systems) may require security level restriction (e.g., SDE(A1, TCB activated trusted path) = B3).[27] Second, this scheme does not only have to be used in conjunction with a security degradation. It can be used to dynamically restrict security levels in any multilevel mode system, for whatever purpose.

MEANINGLESS SECURITY DEGRADATIONS

There are several security degradations in Section 3.8.2, which are very hard to justify as plausible. To start with, all security degradations due to a significant or complete loss of security and/or Biba's integrity labels are not possible (i.e., part of 1, 3a, 3b, and part of 4a). Since this information is part of a secure computers system's critical state, it is the job of the fault tolerance mechanisms to protect it. If loss of this type of information occurs anyway, then it will be nearly impossible to reconfigure the system such that no security or Biba's integrity compromise can occur. This actually pertains to a total loss of any part of the critical state (see Section 3.3).

Another meaningless security degradation occurs in the context of a computer network. The loss of all trusted path capabilities could be handled in a single computer as a multilevel mode degrading to a controlled mode of operation. However, loss of the trusted channel, for a significant period of time, between the network access control center (ACC) and its key distribution center (KDC) (i.e., special nodes in the network), for example, should prevent any multilevel secure communications. This is because the ACC and KDC are part of the network trusted computer base (NTCB) [DoD87]. Breaking apart the NTCB should force degradation into, a minimum, single level C division system.

---

[27]This is not meant as a panacea. For example, 2d from Section 3.8.2 - loss of specifying which user cannot have access to an object, does not warrant this action.

CHAPTER 4

SECURITY PROBLEMS IN ADDING DEPENDABILITY GOALS TO SECURE COMPUTER SYSTEM DESIGN REQUIREMENTS

This chapter explores what has been termed "secure fault tolerance" [Turn86]. Essentially, this involves ensuring that the fault tolerance techniques used in a fault-tolerant, secure computer design do not accidentally or deliberately violate its security policy. The impact that fault tolerance and computer security concerns have on each other is discussed. As an example, the first section of this chapter, testability versus tamperproofness, reveals how a simple fault tolerance requirement, if not carefully integrated with a secure system's design, can lead to a decrease in the trust that can be placed in a design.

Perhaps the most obvious impact of fault tolerance on a secure system design is one of increased complexity [Turn86]. This can be a significant problem for current computer security technology. This is due to the decreased effectiveness of the fault avoidance techniques employed by computer security technology as the complexity of the design increases.

4.1 TESTABILITY VERSUS TAMPERPROOFNESS

The reference monitor [DoD85a] of a secure computer system (which is part of the TCB) has a fundamental requirement to be tamperproof. In order to guarantee this, the LOCK program is implementing the entire reference monitor in hardware [Boeb85a] [Boeb85b] [Boeb85c] [Sayd87]. Additionally, all secure computer systems from classes C1 through to A1 require some hardware and/or software features that periodically validate the proper operation of the hardware and firmware elements of the TCB (i.e., diagnostic capabilities, see Section 3.1) [DoD85a].

In order to provide effective diagnostic capabilities (i.e., with reasonable coverage), hardware should be designed to be testable [Will73] [Funa78]. This facilitates the location of hardware faults so that a failed hardware unit can be replaced or switched out. This is viewed as so important that all IBM mainframes, starting with the early 360 series, include extensive testability mechanisms as hardware standards.

Testability involves the observation and control of a computer system's state. Testability can be utilized for run-time diagnostics or in initial hardware checkout for manufacturing defects. A commonly used technique for testable processor design is Level Sensitive Scan Design (LSSD) [Stol79]. Here, a set of registers in a processor can be connected into one long shift register. Observation of system state is accomplished by shifting all register contents out, and control is accomplished by shifting in test patterns in order to force errors to manifest themselves. This mode of operation is invoked upon the detection of an error.

The issue here is obvious: how to allow the most trusted part of a secure system to be tested, by a technique like LSSD, without compromising its ability to maintain the security of the system? Obviously, if a reference monitor's state is subject to unauthorized modification via LSSD, then it is not tamperproof. The following is a discussion of this issue with LSSD specifically in mind.

After the testing mode is entered, the current state in the affected area of a processor[1] is typically read out and stored for later analysis. For any secure computer system such state information will likely be sensitive. Therefore, its transfer and storage must be protected against unauthorized access (i.e., by analogy, if deliberate hardware design faults are at work, then treat this state information as audit data).

Control of a testable circuit via LSSD involves placing test patterns into the shift register. This implies the need for two distinct modes of hardware operation: test and normal. If this is not enforced adequately, then the reference monitor could be subverted by using LSSD to place it into an insecure state. For example, changing copies of access rights temporarily stored in a processor's register while data is being accessed (e.g., the Intel 286 [Inte83] has segment registers that cache access information when a segment is in use). Also, changing the program counter while in privileged mode can lead to untrusted code running with the privileges of the reference monitor.

Therefore, hardware used to implement testability, hardware used to detect errors that initiate a test mode, and any diagnostic software are targets for malicious logic insertion. Formal verification of both hardware and software is typically used against this type of threat, thereby including all logic associated with testability as part of the TCB (i.e., making it <u>trusted</u> not to be malicious). Following one of the main goals of this study, this reliance on <u>trust</u> should be eliminated. Thus, fault tolerance techniques should be used along with security fault avoidance techniques [Dobs86] to reduce the risk of hardware testability features being used to penetrate a secure system. One

---

[1]Note, that if one failed hardware component in a system enters test mode this does not necessarily mean that the whole computer system stops providing service. Many designs, such as the IBM 3081, isolate the regions of hardware to be tested from the rest of the system.

final note, any other solution to the above problems, that includes the absence of all testability features in a computer system, is unacceptable from a fault tolerance and availability perspective.

4.2 POSSIBLE VIOLATIONS OF A SECURITY POLICY BY FAULT TOLERANCE MECHANISMS

How can the added mechanisms to a computer system for fault tolerance result in compromise and/or denial-of-service? In designing a fault-tolerant, secure system each new mechanism must be evaluated for its potential threat to system security. This section discusses the following standard security concerns. Do any subjects (i.e., here fault tolerance mechanisms) access multiple levels of sensitive data? Can fault tolerance mechanisms be used as covert channels? What can malicious logic in fault tolerance mechanisms do?

4.2.1 ACCESSING MULTIPLE LEVELS OF DATA

Memory scrubbing is a standard fault tolerance technique used to prevent the accumulation of latent errors in memory. It periodically reads every memory location in order to force error correcting codes to fix errors.[2] Otherwise, if a memory word is not accessed frequently enough, then two or more errors can occur in the same word preventing error correction and perhaps resulting in loss of data with a memory failure.

In a MLS system, such a memory scrubber would likely access memory modules that contain data at multiple levels of sensitivity. If it is able to

[2]Error correcting codes for memory can only detect an error when a memory word is read out.

determine the security levels of this data, then it could violate the *-property by use of a covert channel. Placing a memory scrubber entirely in hardware [Renn86] should make it much more difficult—though not impossible—to cause a compromise.

Recovery logic can similarly access multiple levels of sensitive data. First, switching in a cold spare (e.g., for a processor) may include bringing it up to date with the computation that was running before the error. This can involve the transfer of an entire processor's state to a cold spare. Thus, since the recovery logic will supervise this action it can access any of this data.

Second, recovery logic will receive health and status information from every processor under its care. Again in a MLS system, either each processor will be of a different security level or contain a range of levels. Thus, this information will also be at several levels [Turn86]. Third, fault location diagnostics often can use special hardware maintenance channels to examine the state of a malfunctioning subsystem (processor) from another subsystem (processor). If these subsystems contain different security ranges, then this communication channel can result in access to multiples levels of sensitive data.

4.2.2 FAULT TOLERANCE MECHANISMS INTRODUCING NEW COVERT CHANNELS

The added complexity due to fault tolerance mechanisms is likely to result in new information flows and covert channels [Turn86]. Here we present unique instances of both that can be <u>directly attributed</u> to fault tolerance techniques.

Several new information flows are: (a) error reporting from error detection logic to recovery logic (this can be hardware or software), (b) health and status information including diagnostic results (e.g., as in AOSP [RADC85] and

Tandem computers in the form of interprocess messages), and (c) reconfiguration and state restoration control signals from recovery logic to redundant hardware and state. One aspect determining if these information flows could be used for covert channels, is whether or not system defined and protected data objects are used for communication [Kemm83]. Low-level hardware signals can and are used in most designs, and these are not included under the security mechanisms' control (e.g., serially transmitted error reporting messages are too low-level to be audited).

When analyzing computer systems for covert channels the following issues must be addressed: bandwidth of the channel, natural causes of channel noise, type of channel: storage or timing, its ease of use, determining the necessary shared resource used as the channel (i.e., without a shared resource no channel is possible), determining whether the sending and receiving subjects are from different security domains (i.e., different subjects at different security levels, otherwise no channel exists),[3] determine how the sender uses the shared resource to transmit information, how the receiver <u>senses</u> the information sent, and how the sender and receiver synchronizes the information transfer [Lamp73] [Scha77] [Kemm83] [Loep85]. The following subsections present several obvious examples of covert channels due to fault tolerance mechanisms. These are not meant to be a complete list nor the most likely used channels.

Covert Channel 1 – Storage Channel

Figure 4.1, shows a simple N-Modular Redundant (NMR) computation con–

[3]We are currently ignoring discretionary covert channels.

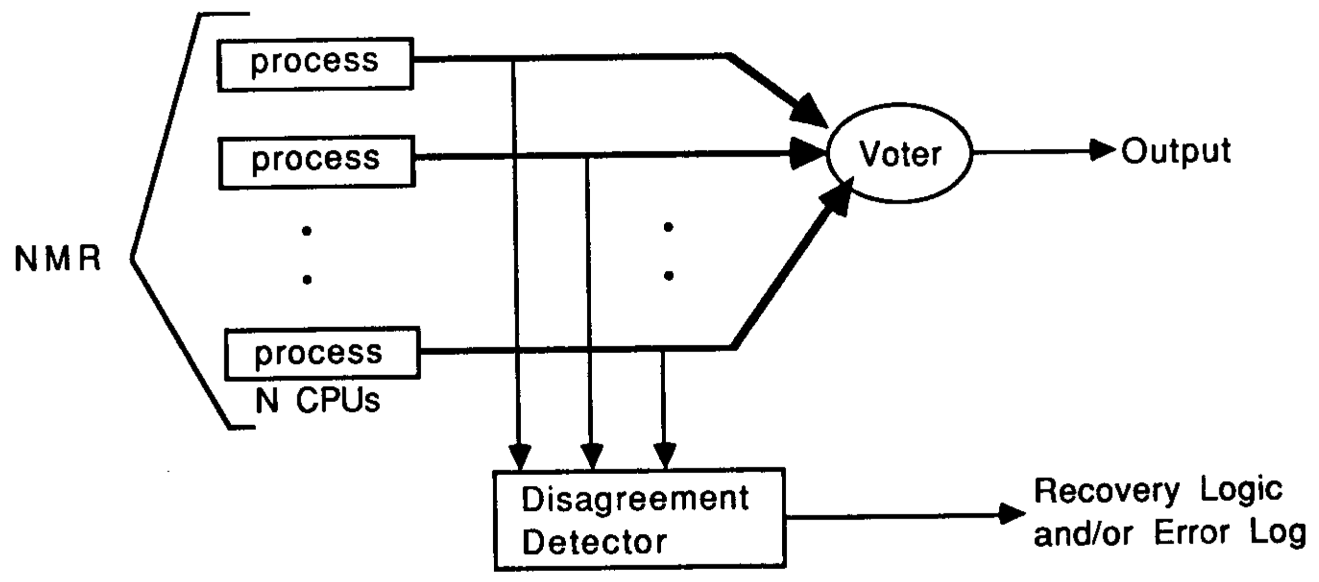


Figure 4.1 Covert Channel 1 - Storage Channel

N-Modular Redundancy (NMR) with disagreement detector [Kueh69] used to determine which CPU(s) has generated an error. A covert channel exists between the processes and the recovery logic and/or any subject that can read the error log.

figuration. Each processor (i.e., CPU) contains the same process (subject). The voter compares the outputs of all processes at predetermined times, and the majority of all values is the output result. If any processor(s) disagrees with the majority decision, then it is recorded by the disagreement detector [Kueh69] and this information is sent to some recovery logic and/or error log.

Thus, the shared resource for this storage channel is whether a processor produces a correct or incorrect result (i.e., the correctness state). The sender is a process and the receiver is the recovery logic or any subject that can read the error log. The storage variable, which is indirectly written by the sender, is in the recovery logic and/or error log which is directly written by the disagreement detector.

The following is the scenario for sending information on this channel. First, one or more processors force a disagreement, by the process running on it, deliberately performing an incorrect calculation. The disagreement detector will detect this and give the processor number(s) to the recovery logic. Obviously, the number of deliberately incorrect processors must be a minority. There are many ways to encode information in this way: (a) each processor may have a value associated to it, or (b) for a 5MR configuration, for example, a 2 out of 5 and/or a 1 out of 5 code can be used to encode many values. The more redundancy used in an NMR configuration, the more information that can be encoded in a single covert transmission.

Each process in Figure 4.1 is actually identical. To send a code by the above method they must produce errors at different times and in different patterns. This can be done by distinguishing each version by a call to a random number generator in each version. This number can then be used to determine when a version should produce an error.

Additionally, the receiver must know the relative order of the processes, determined by this random number, in order to relate their failure patterns to a signalling code value. This can easily be done by a calibration sequence, where for the first few covert transmissions the sender runs through the entire code in some predetermined order.

The bandwidth of this channel can be very high, since it does not interrupt normal system functioning and the recovery logic is the receiver. Thus, the recovery logic can ignore the generated errors. This last point is important, because permanent faults are distinguished from transient faults by their persistence over a time interval. If the recovery logic was not the receiver or at least a "cooperating scheduler,"[4] then the covert channel would force many permanent faults to appear likely closing off the channel and raising suspicion.

Channel noise will occur naturally due to real errors that force some of the processes to occasionally produce an incorrect output when they are not supposed to. This reduction in bandwidth is dependent on: (a) the rate of occurrence of faults, and (b) the probability that faults occur in processors that are not supposed to fail, versus hitting processors that are already supposed to fail.

If real errors should cause degradation of the number of processors in the

---

[4]A "cooperating scheduler," as described in [Scha77], is a scheduler that a subverter [Myer80] has written to help facilitate the transmission of information through the resource that it schedules. This is in the context where the sender and receiver are both processes.

Here we use this concept in a more general form, since a fault tolerance mechanism as a sender or receiver does not have to be a process. Thus, it is used here as any mechanism with some control authority (e.g., recovery logic), which a subverter built, that facilitates the transmission of information through a covert channel.

redundant configuration, then an extra channel used to synchronize the sender and receiver can be used by the recovery logic to inform the sender of this condition. There is no way for the sender to notice a degradation by itself, and thus, the channel would be closed due to noise. So upon receipt of the degradation information, the sending processes can recover by using a simpler signalling code with lower bandwidth.

The sending processes can be at any security level, and over time will change as new processes run on the processors. The recovery logic and any subject that can read an error log will thus, at some time, be at a different security level than the sending process. Additionally, this is not an overt channel of the following reasons: (a) the disagreement detector is not meant as a communication path between a process and the recovery logic [Scha77], and (b) the storage cell is not a system protected data object, and is not normally viewed as a data container [Kemm83]. Thus the above example is a valid covert channel.

Covert Channel 2 – Storage and Timing Channel

Use of a watchdog processor that utilizes derived signatures or assigned tokens as control flow markers [Mahm88] (see Section 3.4.2 on computer viruses), adds both a storage and timing channel to a computer design. The sending process is the program running on the normal CPU being monitored by the watchdog. The receiving subject is the watchdog itself. The main idea, is that the sending process can transfer information by modifying its control flow at specific times. The watchdog will easily detect this by recognizing the generated signatures.

The following is the scenario for sending information on the storage channel. Special loops (i.e., code sequences) are associated with a value. The correspondence between the signature generated by the code loops and the signalling code is done as in covert channel 1 above, that is, via a calibration sequence. To send a code number a process simply runs a special loop. The receiving watchdog processor dynamically generates the signature and compares it with its known signalling code to signature relation.

The following is the scenario for sending information on the timing channel variant. The only difference from the storage channel is that the receiving watchdog senses the information transfer by the number of times that a particular signature is executed in a row, length of time it takes to receive the next signature (i.e., the actual length of the loop), or by the order of a sequence of signatures generated [Scha77].

In order to accomplish the signalling of information via this channel, the extra executions of signaling code loops must not affect the final result of a process. These signalling code loops can be routines that perform some check function, such as reading through a link list to find a special entry. Or, they can be specially implanted code that perform useless functions, and have no effect on the program's results (e.g., extra, useless variables are added to a program and functions are performed on them).

Bandwidth can be restricted if a process has a time limit on it, since the use of the covert channel incurs added execution time. Additionally, as with all the channels presented in this section, noise can occur due to naturally occurring errors.

This covert channel is unique in that it does not require the recovery logic to be a cooperating scheduler, since information is not transferred by forcing errors. This channel is very easy to use, but as just indicated may be detected by changes in execution time [Denn86]. The justification for why this is indeed a covert channel is essentially the same as covert channel 1 above, and is the same for all following channels presented.

Covert Channel 3 – Timing Channel

This channel involves the degradation and reinsertion of hardware resources by the recovery logic. The receiving processes rely on multiple hardware resources of some type(s), such as, memories and processors in order to perform their job. It injects an error the same way as presented for covert channel 1 above, which activates the recovery logic. Then the recovery logic can send information to a process by changing the hardware resources that are available to it by unnecessary degradation and reinsertion of previously degraded resources.

The change in performance observed by a process encodes the transmitted information. The bandwidth of this channel is limited by the overhead of the recovery logic in degrading and reinserting hardware resources, and again, by the occurrence of real errors forcing degradation. The channel is easy to use, but can be detected by watching the frequency of the change of state of redundant resources.

Covert Channel 4 – Timing Channel

This timing channel involves the delaying of a process by the use of diagnostic procedures which are periodically run on the entire computer system. The sender is the diagnostic routine and the receiver is a process. The shared resources are simply any hardware resources that a process uses, such as processors, memory, and input/output devices.

The transfer of information can occur either by the order in which the diagnostics are performed, or by the length of time they take. Diagnostics can only be used as a transmission media if they are performed in the background while other hardware resources are still working on an application. This results in an application waiting until a particular diagnostic is over if it needs the hardware resource under test.

The bandwidth of this channel is reduced by these factors: (a) it can only be used when the diagnostics are run, and (b) diagnostics have a reasonable limit on their execution time. Thus, if either of the two above properties are violated, this channel will be easy to detect.

4.2.3 MALICIOUS LOGIC IN FAULT TOLERANCE MECHANISMS

The contents of this section are rather straightforward. Fault tolerance mechanisms tend to have <u>very significant</u> control over the resources (e.g., memories, application tasks), and data of a fault-tolerant computer. This degree of control is typically deemed necessary in order for it to be able to recover the system in complex fault conditions.

Malicious logic in the error detection and/or recovery mechanisms can lead to both compromise and denial-of-service. Several general examples of this are: (a) error detection can force a high false alarm rate resulting in denial-of-service [Turn86], (b) either detection or recovery mechanisms can ignore errors, thus likely leading to improper service and denial-of-service, and (c) the error recovery mechanism could maliciously perform its recovery task. There are many examples of malicious error recovery. First, error recovery could place the secure system into an insecure state by changing the state of subjects' privileges. For example, in the Intel 286 [Inte83], the segment tables define a subject's access to objects. Recovery of such tables, due to some accidental or deliberate error, could include unauthorized adding or removing of subjects' privileges. Second, in switching in a cold spare memory module, data is typically copied from the working module(s), as such, data of different classifications could be intermixed. Lastly, recovery logic could perform unnecessary degradation resulting in denial-of-service.

With this degree of power, computer security technology would place much of the fault tolerance mechanisms into a secure system's TCB. The next section discusses the problems with attempting to do this.

4.3 THE IMPACT OF FAULT TOLERANCE ON THE SECURITY DESIGN

The use of purely fault avoidance techniques to build a TCB results in two effects: (1) it limits the TCB's functionality, since it must be small and simple enough for the off-line techniques to be effective, and (2) it has no run-time protection from the unavoidably remaining design faults nor any physical faults [Dobs86].

Thus, the most obvious impact of fault tolerance concerns is the clash of views on the use of fault avoidance techniques. The fault tolerant design will include the possibility of faults anywhere in the system. Thus, the entire TCB will have to be able to tolerate a defined set of faults, including unintentional and deliberate design faults. The use of additional mechanisms to tolerate design faults (both hardware and software) remaining in the TCB is clearly in clash with the current computer security design approach. First, from the security viewpoint, what assurances will be provided that these additional mechanisms will not violate the security policy? Second, the fault avoidance techniques relied upon are supposed to provide enough assurances of the absence of design flaws (faults).

The first point is a very valid concern. Examples of how fault tolerance mechanisms can violate a security policy were presented in the last section. The use of NVP previously in this dissertation is, however, an example of a fault tolerance technique that polices itself (also see Chapter 5). The additional versions cannot violate the security policy, and it is just the support software and hardware that must either be diverse or evaluated using security fault avoidance techniques. Thus, all fault tolerance mechanisms that are used in a fault-tolerant, secure computer, must be designed to tolerate accidental and deliberate design faults! However, the second point is not valid, and should be handled by a judicious use of both fault avoidance and fault tolerance techniques.

Placing most of the fault tolerance mechanisms of a fault-tolerant, secure computer design into the TCB, as was suggested in the last section, and adding fault tolerance techniques to make the TCB itself fault-tolerant will present a major difficulty for current computer security technology. All the fault tolerance mechanisms in the TCB would have to be proven to correctly implement their design specifications in both the absence and presence of faults. Also, the design would have to be proven to be fail-safe (i.e., a fail-secure design [Molh73]). The prospect of effectively applying computer security technology's formal verification and certification techniques to such a TCB is bleak. First, error recovery mechanisms work in a very large state space[5] with potentially several (different) recovery mechanisms working in parallel. Second, error

[5]Recovery logic has to deal with many different types of errors, degrees of error propagation, locations of errors, and number of errors.

detection and recovery will be asynchronous to normal system functioning. This makes formal verification, simulation, and testing infeasible under all fault conditions for which the system is designed.

As an indication of the difficulty of this, current techniques used in fault tolerance to assess the effectiveness (i.e., coverage) of fault tolerance mechanisms (i.e., fault and error injection into hardware and simulations) have difficulty in deriving believable numbers, and are extremely difficult to apply. Also, detection and recovery mechanisms used may be many, and of several different types.

Nevertheless, there exists a precedent for the application of formal methods for verification of a fault-tolerant computer design. The SIFT (Software Implemented Fault Tolerance) project at SRI applied a formal specification and verification approach to aid the evaluation of the computer system's reliability [Wens78] [Gold80] [Mell82].

An actual <u>code</u> level proof of the small SIFT operating system (less than 1,000 lines of Pascal [Gold80]), plus a standard hierarchy of formal specifications was employed for system design verification. An interesting aspect of this study was the association of both reliability and error rate Markov models to the formal specification hierarchy. This was done in order to include in the formal specification the probability that the system had enough working parts to support the functional aspects modeled.

The verification process on the SIFT design did result in uncovering four significant design errors. Whereas this work is fundamentally important, the SIFT computer system has a simple design, thus allowing a fairly complete verification. This is not typical of many other computer systems.

4.4 THE IMPACT OF SECURITY PRINCIPLES ON THE FAULT TOLERANCE DESIGN

A basic principle in dependable computer design is that a reliable computer can be built from unreliable parts (or that a reliable distributed system can be built from unreliable nodes [Dobs86]). However, this does not extend to secure system design [Neum86]. The following observation should help clarify this. Fault tolerance techniques are mainly concerned with preventing errors from propagating out to a service boundary, thus causing a failure. However, computer security techniques are instead mainly concerned with preventing subjects from obtaining unauthorized privileges, and thus, breaking into the system.

If a secure system were composed of layers of insecure parts as suggested in [Dobs86], then all an attacker needs to do is to penetrate the lowest insecure layer in order to overcome the entire system [Paan83] [Neum86, p.908]. This was one motivation behind the creation of the TCB concept. Current computer security technology builds secure distributed systems from secure parts [Fell87].

Thus, the view that distributed secure systems can be built from insecure parts, as presented in [Dobs86], is flawed. We propose a design of a fault-tolerant, secure system (i.e., a single computer or a network of computers), that follows basic security principles (i.e., it still relies on a TCB). However, in order to alleviate some of the above mentioned deficiencies of TCBs, fault tolerance techniques are used to make the TCB the hard core of the system.[6]

One should note, that this design in itself is not effective against the

[6]A TCB has always been the hard core of a secure system, but now we advocate using fault tolerance techniques along with fault avoidance techniques to achieve it in the context of a fault-tolerant, secure design.

denial–of–service threat. For this threat, techniques presented throughout this dissertation should be applied throughout the entire computer system. That is, to all software and hardware, where appropriate, since no boundary can be drawn to deal with this threat [Glig85a].

A general design of a fault–tolerant, secure computer system contains the following properties. (1) A TCB (or one for each computer in a network [Fell87]) is used to address standard security and integrity policies. (2) Fault tolerance techniques are used for software and hardware to prevent the effects of malicious logic, and physically occurring faults (both accidental and deliberate).[7] Reliance on trust that these mechanisms will not violate a security policy must be reduced. Thus, whenever possible, fault tolerance techniques that police themselves should be used (see the last section). (3) A fail–safe mechanism(s) is used so that if the hard core TCB fails, then the entire system stops with minimal damage and compromise (i.e., a fail–secure design [Molh73]). Since the TCB is the hard core, if it fails to provide security from outside attacks and from malicious logic contained in itself, then the system will fail entirely.

This design approach is compatible with all design options presented in Section 3.5. However, option C–allow security relevant errors requires a new definition for compromise in a fault–tolerant, secure computer system.

In [Hsia79] [Turn86] [RADC87], it is stated that state redundancy (e.g., checkpoints used in backward error recovery) increases the exposure potential of sensitive information. However, this can be reduced to an acceptable level

[7]This does not mean that every piece of software runs as an N–version system. See Section 2.5 on tradeoffs.

by the use of fragmentation–scattering. Additionally, to prevent unauthorized modification of information and programs, data redundancy will make it harder for a malicious subject to corrupt all valuable data. This holds because it will take more modifications to corrupt data, and this increased activity may be easier to detect.

CHAPTER 5

EXPERIMENTATION AND EVALUATION

Effectiveness measurements of the countermeasures chosen can be used in design refinement. If serious protection problems are discovered during measurement, steps can be made to compensate.

To obtain this measure, deliberate insertion of malicious logic similar to the technique of fault and error injection used in fault tolerance to determine fault coverage [Schu87], and error seeding techniques used in software testing [Rama81], can be employed. The determinations of what malicious logic should be used and where to implant it can be made in several ways. First, use a penetration team whose experience in security concerns helps them devise implants. Second, test specific conditions addressed by existing on-line security mechanisms. And last, use analysis techniques such as fault-tree analysis as employed in software safety [Leve85] [Leve86].

Deliberate insertion of malicious logic for testing purposes obviously must be done with great care. The process of implanting must be well documented and performed by a team. Implants should be placed only in experimental versions used solely for testing. These versions should never be placed in the same configuration library where the real operational software is stored. Otherwise, some malicious logic used for testing may be deliberately left in an operational system.

Measurements of effectiveness of both on-line and off-line techniques can be performed. For on-line techniques, malicious logic is placed in operating software during normal system testing. For off-line techniques, such as formal verification, malicious logic is deliberately placed in preverified code during early design stages.

Performing such measurements requires defining what is to be measured (the metric) and how the results are to be evaluated (the criterion). The metric is the percentage of instances, where implanted malicious logic goes undetected out of the entire body of tests [Schu87]. The criterion is composed of two parts. First, it must take into account the coverage, or quality of the error seeding cases [Rama81]. Second, the calculated percentage is interpreted relative to the acceptable degree of risk defined in the design phase [Jose87]. Two results should be generated: one for all on-line, and one for all off-line techniques used. This way, the return on investment of each approach can be compared.

5.1 N-VERSION PROGRAMMING (NVP) USED FOR SECURITY

Security is used here in relation to the prevention of the effects of malicious logic on computer systems. This section is a discussion of the general issues in using NVP for security.

To overcome the use of NVP, malicious logic must be able to cause a majority of coincident errors at a voting point(s) (multiple malicious actions may be necessary to do real harm). If the coincident errors are similar, then the result would be a loss of integrity of function and data (i.e., deliberate improper service), since the voter (through consensus decision) would accept malicious actions. However, if the coincident errors are dissimilar, then the result would very likely be denial-of-service, since NVP systems typically do nothing without a majority decision (also see Section 2.3.1).

5.1.1 ACTION VOTING

A deliberate design fault (DDF) in just one version could easily circumvent most NVP decision mechanisms. A DDF, for example a Trojan horse, would perform devious actions (e.g., delete necessary files) in the background, while producing valid data results to be voted on. To assure defenses against such a threat, "action voting" (as introduced in [Chen78]) is needed (i.e., actions taken by a version are voted on, as well as, the data generated). Thus, if less than a majority of versions try to perform an action at the same time, then a malicious attempt will be masked out.

The actions (e.g., system function calls) that need to be voted on, are external manipulations of system state. An editor with an embedded Trojan horse may attempt to make an extra copy of a file it is editing for the Trojan horse's author (a classic example). This unauthorized copying is an external action, since it leaks information to another subject. Internal version actions (e.g., call to an internal sort routine) cannot directly cause malicious changes to the system state. Thus, since internal actions do not immediately cause malicious results, and in order to reduce the possible performance impact that extensive voting may incur, only external version actions should be voted on. Note, preventing all unspecified external actions is part of the standard NVP design concept.

5.1.2 THE MONITOR MUST DIRECTLY MAKE ALL VOTED ACTIONS

Of still greater importance, is that all actions that will be voted on must only be made by the monitor (e.g., DEDIX [Aviz85b]). The reason for this is two fold: (a) this will prevent a malicious version from lying about what action it will perform (i.e., for voting it advertises that it wants to perform action a_1, but after a successful vote it performs action a_2 instead, which is a malicious action), and (b) a version must be prevented from performing an external action any time it wants to (i.e., at a non-voting point). This can be implemented by requiring the monitor (i.e., the operating system) to only perform a function that has been favorably voted on by the majority of the versions. This is easily accomplished since the voting mechanism is part of the monitor. Thus, for NVP-based programs the interface for executing external actions is designed to be through the monitor.

5.1.3 RESTRICTING A VERSION'S EXECUTION SEQUENCE

In order to vote on generated data items, current NVP-based systems restrict, to some degree, the general sequence of execution and voting points (i.e., cc-points) [Kell86] [Aviz87b]. In fact, in a recent experiment using NVP for an airplane's automatic landing system prototype [Aviz87b], only one basic execution sequence was plausible. As an analogy, in order to make action voting manageable, the execution sequence of external actions should also be dictated. To prevent excessive voting points, data and action voting should be done at the same time whenever possible.

Restricting the execution sequence naturally forces the question: "How does this effect the diversity of the versions?" If we are designing a fault-tolerant, secure computer system, then this question is of great concern. However, if NVP is used solely to prevent the effects of malicious logic, then loss in diversity is not important, since it will still prevent malicious actions by a minority of versions regardless. In two NVP experiments at UCLA [Kell86] [Aviz87b], restriction of execution sequence did not cause a large decrease in diversity.

5.1.4 ATTACKS ON NVP-BASED SECURE SYSTEMS

One criticism that has already been made of an NVP-based system involves a classical form of denial-of-service. If one of the versions deliberately delays its execution in order to slow down the processing of the entire computer system, then the required performance could be denied.

This threat is simply not possible, since all voting points are timed. This timing delay attack is a classical fault case [Kope85] [Ezhi86], and is easily handled in NVP-based systems. Thus, if one version slows down its execution and does not reach the proper voting point on time, it will then be marked as failed and the NVP-based system will continue without it. NVP systems are being used in real-time applications, such as the flight control system of the European designed Airbus 320 [Rouq86] [Avia87].

However, NVP-based systems are vulnerable to at least two types of attacks: computer virus errors (see Figures 3.1 and 3.2), and malicious logic in support hardware and software. A computer virus could infect all or just a majority of the versions, resulting in malicious actions being accepted by the voter. Additionally, the NVP software monitor (i.e., the executive) could also be a target for viral infection.

In regard to both of these types of attacks, it should be obvious, that a fault-tolerant, secure computer must ensure that its support mechanisms are protected from accidental and deliberate faults (also see Section 2.3.1, sub-section on background devious actions, and Section 3.6).

5.2 IMPLANTING MALICIOUS LOGIC IN NVP SYSTEMS

To explore the ability of NVP to resist DDFs, injections of DDFs into an NVP-based application were performed. This experiment was designed to gather insights on the difficulties, if any, of forcing an NVP-based system to fail in the case when an attacker could modify a majority of the versions. Please note, that the assumption made in Chapter 2 was that an attacker (i.e., a malicious engineer) could only modify one version. We relaxed this constraint in order to explore the effect of design diversity on the effectiveness of implanted DDFs. This experiment used no off-line techniques to detect malicious logic, but the injections were performed with the constraint that malicious logic should be hard to find by such techniques.

5.2.1 ASSUMPTIONS, LIMITATIONS, AND ORGANIZATION OF EXPERIMENT

It was assumed that the best method of hiding malicious logic entailed its insertion into the complex parts of a version. Such an implant would be designed so that the malicious logic was essentially invisible to code inspection (see Section 5.2.5).

The malicious logic injection experiment was constrained by several limitations. First, no action voting was done. This occurred because the mechanism to ensure that only the monitor could perform an action (see Section 5.1.2) was not available. Thus, injected malicious logic was designed only to affect data generated by the N-version experimental testbed (see Section 5.2.2).

Second, only one form of malicious logic was injected, that being, DDFs. Malicious logic can also take the form of specification faults, algorithm faults, and computer virus errors (see Section 2.2). Injection of the other types of malicious faults were not possible, since the experimental N-version system was already designed and running.

Third, and last, enabler components of DDFs (see Sections 2.4.3 and 5.2.3) were also not injected. Enablers require a computer system to have a user interface, that allows human interaction with the system, in order for it to be useful to DDFs. However, the N-version system used had no such interface.

The malicious logic injection experiment was organized into two parts. Part 1, as described in Section 5.2.3, involved DDF injection in order to determine the difficulty of causing coincident errors, and while doing so to study their makeup and behavior. Part 2, as described in Section 5.2.4, is aimed at determining if DDFs that involve added functions (i.e., extra, unspecified actions by a version) can be made to cause timing faults, and thus, be detected at an NVP system's voting point.

ANALYSIS CONDUCTED BEFORE INJECTION

The design of, and locations to inject DDFs was done by a careful and thorough procedure. An extensive, non-computer based analysis of the design, and implementation of each version of the NVP testbed was conducted. This involved the following steps:

(a) familiarization with the specification of the application software composing the NVP-based testbed,

(b) code inspection of each version, and flow charting of parts of each version in order to determine its structure and locations of complexity,[1] and

(c) fault trees were employed, so that working from the error and its desired location of manifestation, all places where a fault could cause the error were derived. Locations to inject DDFs were finally derived by using these fault trees, and the complexity profile of each version derived in "b" above.

In order to derive DDFs that were essentially invisible to code inspection, the uniqueness of each version was taken into account. Each version's style of coding and formatting were imitated, and common faults (bugs) that occur in its implementation language (see Section 5.2.2) were exploited. Additionally, each version's type of complexity was also imitated.

5.2.2 THE EXPERIMENTAL SYSTEM

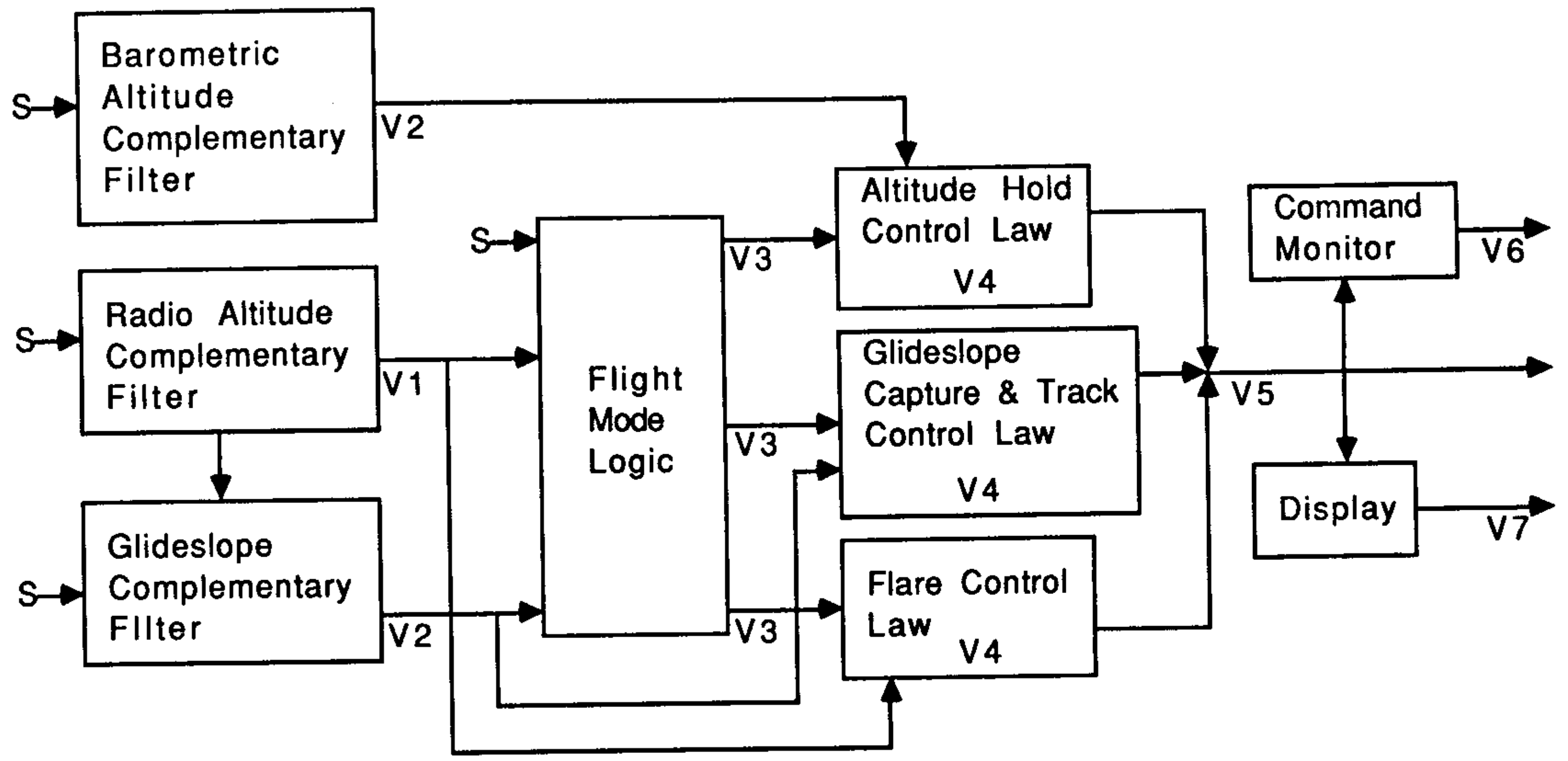
The testbed for the malicious logic injection experiment was an NVP-based system for the automatic (i.e., computer-controlled) landing software for a commercial airliner (from here on we will refer to this as "Autoland"). The N versions were developed as part of another research project at UCLA, that focused on exploring issues in N-version software [Aviz87b].

The NVP-based Autoland system was originally developed with 6 diverse versions, in 6 different programming languages (namely, Ada, Pascal, C, Modula-2, T [a dialect of Lisp], and Prolog). For the malicious logic injections, only the Ada, Pascal, C, and Modula-2 versions were used.

The basic structure of one version of the Autoland system appears in Figure 5.1. The locations of all the voting points used in N-version operation and the important data flow paths are indicated.

During Autoland operation pitch (vertical motion) modes entered in the landing process are: Altitude Hold, Glideslope Capture and Track, Flare, and Touchdown. Flight mode entry and exit is determined by the Mode Logic equations, which use filtered airplane sensor data to switch the controlling

[1]What comprises program complexity is difficult to define. However, it can be described by example: tricky and hard to understand coding practices, hard to understand algorithms, poor program format (i.e., fails to indicate program semantics), lack of meaningful comments and variable names, dense code, many variables used as flags for conditionals, large run-on procedures, etc. Complexity can be qualitatively and quantitatively expressed (e.g., number of IF statements in one procedure).



Barometric Altitude Complementary Filter
Radio Altitude Complementary Filter
Glideslope Complementary Filter
S
S
S
S
V2
V1
V2
Flight Mode Logic
V3
V3
V3
Altitude Hold Control Law V4
Glideslope Capture & Track Control Law V4
Flare Control Law V4
V5
Command Monitor
Display
V6
V7

Figure 5.1 Functional Components of the Autoland Test Versions

KEY: S - sensor inputs, V1 - voting point for filter 1, V2 - voting point for filters 2, V3 - voting point for flight mode calculation, V4 - voting point for outer loop of control law, V5 - voting point for inner loop of control law, V6 - voting point for elevator command, V7 - voting point for autoland display.

equations at the correct point in the trajectory.

Flight begins with the initialization of the system in the Altitude Hold Flight mode, at a point approximately ten miles from the airport. Initial altitude is about 1500 feet, initial speed 120 knots, with zero flight path angle. Responding to turbulence–induced errors in altitude and altitude with automatic elevator control motion, the aircraft maintains the reference altitude until the edge of the glideslope beam is reached (i.e., the Altitude Hold Control Law controls this phase of flight).

If the capture conditions are met, the airplane enters the Glideslope Capture and Track Flight mode and begins a pitching motion to acquire the beam center. A short time after capture, the track mode is engaged to reduce any static displacement towards zero (i.e., the Glideslope Capture and Track Control Law controls this phase of flight).

The airplane maintains a constant speed along the glideslope until an altitude of about 50 feet is reached. Flare logic equations determine the precise altitude at which the Flare flight mode is entered. In response to the Flare Control Law, the vehicle is forced along a path that targets a vertical speed rate of two feet per second at touchdown.

Upon entering Touchdown flight mode (altitude less than 10 feet), the automatic portion of the landing is complete and the system is automatically disengaged by the Flight Mode Logic. This completes the automatic landing flight phase. This Autoland system is designed to run with 2 or more separate processing units each with diverse software.

The Command Monitor, in Figure 5.1, compares locally computed elevator (an airplane's control surface) commands with ones generated by the other versions. This provides the basic fault detection function in critical flight control systems. The Display module provides continuous information to the flight crew on autopilot mode, including the Autoland's fault status.

5.2.3 PART I: FORCING COINCIDENT ERRORS

Implanting well concealed malicious logic into a majority of versions to force similar or dissimilar coincident errors is dependent on two points. First, and obviously, a mole must have access to a majority of the version's source code.

Secondly, a static analysis of the four UCLA versions from the Autoland experiment (i.e., of the program code), indicates that versions have their complexity in a different portion of their design [Schue87] (e.g., the C version is complex in its Flight Mode Logic computation, while the Ada version is complex in its main control loop). However, some similarities do occur (e.g., both the Ada and Pascal versions have complex main control loops).

This analysis leads to the following two observations: (1) to force coincident similar errors may require different faults in each or several of the versions, and (2) similar design faults may be placed in different portions of each version, but that each fault will then generate an error at different times and probably in different system states. This may result in difficulties in triggering and controlling the malicious logic in each version for producing errors at the same voting point.

Injected DDFs were designed to force errors at the following voting points in the NVP-based Autoland system (see Figure 5.1): case 1 – V4 and V5 to force the failure of the Control Laws, case 2 – V2 to force the failure of the Glide-slope Complementary Filter, case 3 – V3 to force the failure of the Flight Mode Logic, and case 4 – at any voting point by forcing improper initialization at flight mode changes. (Note, that mini-initializations are performed as flight modes change in the Autoland application. Faults were designed to force one of these initializations to occur too soon or not at all.)

Tables 5.1 to 5.4 present the number and location of DDFs injected for each case outlined above. In Tables 5.5 to 5.8, the total number of coincident errors (part "a" of the tables), and the number of coincident similar errors (part "b") forced by the injected DDFs are presented. Out of the total number of coincident errors forced, many combinations of coincident dissimilar error exist. In Table 5.6(b), a large number of coincident similar errors is noted. This was due to a common location in the Pascal and C language versions where an absence of diversity was found. Note, this was the only point where such ease in forcing coincident similar errors was found.

These numbers are **not** for the immediate use in deriving coverage estimates (i.e., an effectiveness measure). They are too small a sample and were injected only by one person (i.e., the author). They are to be used only as an indication of the relative difficulty of forcing coincident similar errors in NVP-based systems where a malicious engineer has access to more than one version. Annotated samples of the most interesting injected DDFs are presented in Appendix B (which includes comments about hiding DDFs).

What is of more importance than the above mentioned numbers are the insights gained from this part of the experiment. As a result of the experimental injections it was observed that DDFs have the following general structure:

$$\text{DDF} = [\text{Enabler(s)}:\text{Trigger(s)} \rightarrow \text{Error Producing Logic}].$$

This notation indicates that the enabler component of a DDF enables/disables the triggering device, and the trigger once fired, executes the error producing logic. Note, that all three components make up the fault (e.g., the error pro-

Version	Location of faults	Number of faults
Pascal	-Glideslope Capture and Track Control Law,	6
	Flare Control Law	2
C	-Flight Mode Logic	9
Modula-2	-Display Module	12
Ada	-Main Control Loop & Utility Functions	12

Table 5.1 Case 1 Injected Faults to Fail the Control Laws

Version	Location of faults	Number of faults
Pascal	-Glideslope Complementary Filter	13
C	-Glideslope Complementary Filter	12
Modula-2	-Display Module	9

Table 5.2 Case 2 Injected Faults to Fail the Filters

Version	Location of faults	Number of faults
C	-Flight Mode Logic	10
Modula-2	-Display Module	9
Ada	-Flight Mode Logic	9

Table 5.3 Case 3 Injected Faults to Fail the Flight Mode Logic

Version	Location of faults	Number of faults
Pascal	-Flight Mode Logic*	3
C	-Glideslope Complementary Filter	4

Table 5.4 Case 4 Injected Faults to Force Improper Initialization

*Not justifiably complex to hide DDFs. Was done to observe the diversity of errors produced.

A total of 110 faults were injected.

Note, in each table above, missing versions mean that DDFs could not be injected to cause the specific error, because of insufficient complexity in the version.

Version	Total Coincident Errors by Time			
	T_1	T_2	T_3	T_4
Pascal	3	2	3	0
C	3	1	3	2
Modula-2	1	1	2	3
Ada	4	3	2	2

(a)

Number of Failed Versions	Instances of Similar Coincident Errors
2	3
3	3
4	0

(b)

Table 5.5 Case 1 Coincident Errors

Version	Total Coincident Errors by Time		
	T_2	T_5	†
Pascal	1	4	
C	1	3	
Modula-2	2	4	

(a)

Number of Failed Versions	Instances of Similar Coincident Errors
2	11*
3	1*

(b)

Table 5.6 Case 2 Coincident Errors

Version	Total Coincident Errors by Time		
	T_6	T_7	T_8
C	5	2	1
Modula-2	6	1	1
Ada	6	1	0

(a)

Number of Failed Versions	Instances of Similar Coincident Errors
2	1
3	2*

(b)

Table 5.7 Case 3 Coincident Errors

Version	Total Coincident Errors by Time	
	T_1	T_2
Pascal	1	1
C	2	0

(a)

Number of Failed Versions	Instances of Similar Coincident Errors
2	0

(b)

Table 5.8 Case 4 Coincident Errors

*Different input cases can lead to non-coincident and/or dissimilar errors.
†The rest of the coincident errors are dispersed in time.
T_1 to T_8 represent different real-time values during Autoland flight.

ducing logic, by itself, is not a DDF).[2] This last point is important, since the enabler and trigger have a significant impact on the DDF's behavior.

The trigger component can be designed so that the DDF becomes a stuck-at or intermittent fault (i.e., once turned on it stays on, or it changes state according to some distribution). The trigger can simply be set to fire on system state conditions (e.g., the date, or the number of enemy targets seen), or even be counter based [Myer80].

The enabling component is designed as an input from a system's user. Otherwise, the enabler behaves just as another trigger. For example, the enabler could be a sequence of legal but odd system requests.

The error producing logic component, once executed, can result in an error, either immediately, or after some delay. This delay occurs because some instances of error producing logic must affect a system's state over several iterations in order to force an observable error. The trigger and a delayed error producing logic together allow a DDF to be enabled with a delay, so that the enabling user does not have to be present at system failure.

Devising DDFs that were well hidden was a difficult task. So the question was asked: "Why is it hard to implant malicious logic into an existing NVP system?" The answer is that the degree of complexity required to hide malicious logic is not necessarily present in enough of the versions (e.g., the Modula-2 version was very well built with only one complex part). A designer whose aim it is to implant malicious logic into his/her version would purposely make it complex (e.g., the C version uses all global variables and is very hard to

[2]Computer virus faults and errors are an exception, because they do not need enablers.

understand due to its poor coding style). Placing malicious logic into a version after implementation can involve adding complexity to that version. This may require too many changes to go unnoticed.

The diversity of the location of complexity in each version restricted where and what kind of errors could be produced. Only the Control Laws (voting points V4 and V5) where relatively easy to corrupt. In fact, in case 4 – forcing improper initialization, no coincident errors were produced.

It was observed that similar injected faults could result in different errors. The best example of this was again the DDFs designed to force improper initialization (i.e., case 4). The manifested errors were significantly different in degree, with the Ada version changing most of its variables to zero, while the Pascal version changed only a few key variables.

Synchronization problems did occur with the Modula-2 version, because its implanted DDFs were quite different from the faults in the other versions. It was hard to design triggers so that the versions would cause coincident errors over variable inputs. Well design triggers kept errors coincident, but errors would still appear at different flight times depending on the input.

Overall, coincident similar errors were possible to a limited extent, but required a great deal of work even when all versions were accessible to the malicious engineer. However, coincident dissimilar errors were much easier to force, and could potentially result in denial-of-service.

5.2.4 PARTII: CONSTRAINING MALICIOUS LOGIC WITH TIMING RESTRICTIONS

If all external actions cannot be voted on, due to performance considerations, what can be done to decrease malicious activity of the non-voted actions? Unspecified actions of malicious logic use up extra time, and perhaps, can be detected as timing faults.

Each voting point in an NVP-based system is timed, meaning that every version must provide data to the voter within a time window. If a version misses the time window it is marked as failed and the rest of the NVP system continues on without it. This mechanism exists to detect timing errors. The time used by the extra malicious actions may delay the version's progress and make it miss a voting point.

Thus, in a NVP system design, a tight but workable limit should be set on all voting points, and adding a few extra voting points than are actually necessary is advisable. This will constrain malicious logic into being careful of how much time each action takes, and on the cumulative time of all actions between voting points.

A possible result of the above constraints on a version's execution time may be to slow down the malicious logic's unspecified actions. These harmful actions will probably have to be spread out over its entire execution sequence. This may make them easier to detect, and may even prevent some unspecified actions completely (e.g., mounting a tape to read unauthorized data may simply take too long).

In order to derive tight but workable time limits on voting points, time measurements were performed on the 4 Autoland versions. The estimated times do indicate diversity in execution times throughout the versions. However, during the beginning of DDF injection in this part of the experiment, a DDF was devised that can hide from the proposed timeout detection mechanism.

The basic structure of a DDF, as introduced in Section 5.2.3, is extended to:

$$\begin{aligned}\mathrm{DDF}_T = {} & [\mathrm{not(Enabler(s))} \rightarrow \mathrm{Time\ Consumption\ Logic}\ ]\ \& \\ & [\mathrm{Enabler(s){:}Trigger(s)} \rightarrow \mathrm{Error\ Producing\ Logic}].\end{aligned}$$

The difference from the basic DDF is the first clause, which is designed to consume time equal to, or greater than, the time required for the error producing logic. Note, that the two clauses of the DDF_T do not have to be textually adjacent.

A DDF_T alludes detection as follows. Initially, detection by a timeout mechanism was based on the assumption that malicious logic would not execute during test or during time measurements (these are needed for real-time applications). Thus, the time spent for malicious logic would not be counted. However, DDF_Ts always run, but do not generate errors while the enabler is off. Note, that DDF_Ts are still somewhat constrained by an applications real-time limits. Lastly, the time consumption logic will likely appear as some useful function.

An obvious variant of DDF_T is:

$$\begin{aligned} DDF_T' = [&\text{not(Enabler(s):Trigger(s))} \rightarrow \text{Normal Function,} \\ &\qquad\qquad\qquad \{\text{ else: }\} \quad \text{Error Producing Logic],} \end{aligned}$$

where the time required for the error producing logic is less than, or equal to, the time required for the normal function.

5.2.5 QUESTIONS ABOUT THE INJECTION EXPERIMENT

Two questions concerning the malicious logic injection experiment, described above, have already arisen. These are: "Why can't DDFs be hidden in simplicity, instead of complexity, in order to force coincident similar errors?," and "Can branch coverage analysis be used to detect enablers and triggers?"

The idea of hiding DDFs in a version's simplicity (i.e., a small change that is obvious enough to be overlooked during code inspection), allows identical faults to be implanted in all or a majority of versions to hopefully force identical errors. Some fault injections of this form have already been done on an N-version application.[3] The injected fault was a slight modification to a constant value in all versions (e.g., change 65.44 to 65.22). Under normal operating conditions and many test cases this fault never generated an error. However, during some rare and/or stressful input state all the versions failed identically. (This is an example of each version obtaining the same bad data and was discussed in Section 2.3.1, subsection on background devious actions.) Notice, that the trigger for this type of DDF is implicit (i.e., the rare and/or stressful state). Also, this fault, as described, had no enabler.

For this type of fault to be used deliberately, it would most likely be generalized. Thus, the fault would be a very slight and simply done modification to some calculation (e.g., adding or subtracting a value from a formula or variable).

This type of DDF seems to have several problems. First, since good testing methods try to determine system behavior in rare and/or stressful input conditions the DDF may need an enabler. Thus, this fault may not be as easy to implant as it first appears. Second, it seems that code inspections would catch many of even the simplest faults (i.e., obvious faults have a tendency to jump right out of the page to the reader). Nevertheless, this type of attack seems more feasible than hiding DDFs in the complexity of a system (due to the observations of the experiment), and thus, warrants further study.

Concerning the second question, branch coverage analysis measures the percentage of branches (i.e., conditionals) that have been forced to execute by test cases. If all branches are forced to execute, then all code in a program will be forced to run at least once. Therefore, if all code runs at least once,

[3]Done at North Carolina State University by Dr. Vouk [Private Communication].

then won't all triggers and enablers be found? This argument seems valid at first, but a simple counter is possible.

A trigger (or enabler) can be designed so that it requires an obscure path of program execution to run in order to fire (i.e., turn it on). Thus, since branch coverage analysis, nor <u>any</u> testing technique, can test all paths of computation through a program, the trigger (or enabler) will not be detected. As an example, consider a trigger consisting of the multiplication of two variables: t1 and t2. Each are initialized to zero, and set to one in different <u>IF</u> statements of a program. Thus for the trigger to fire, both <u>IF</u> statements must be executed, and thus, a particular path. The error producing logic could then take the form of a simple extra term added onto a pre-existing and correct equation (see example 6 in Appendix B)—thus not detectable by branch coverage analysis. This extra term would contain the trigger.

Whereas the above example describes a counter to branch analysis, this off-line technique seems useful in detecting obvious triggers and enablers. However, it should be obvious that as a system becomes larger and/or more complex, such an analysis becomes more costly and less effective.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This chapter has been written with the intent to put the dissertation in perspective. Therefore, Section 6.1 outlines what is considered to be the achievements of this research, Section 6.2 discusses where this work can be used, Section 6.3 presents much of the related, previous work that formed a basis for this research, and lastly, Section 6.4 presents several important, challenging, and time consuming directions for future work. These future research directions are viewed as essential in order to move fault-tolerant, secure computing research into engineering practice.

A great deal of ground has been covered by this dissertation, but much remains to be done. Fault-tolerant, secure computing systems have been, and still are a rich area for research.

6.1 CONTRIBUTIONS

In the beginning of this dissertation (Section 1.8), a set of questions from [Turn86], which starts to define this area of research, was introduced. Progress has been made in answering these questions, and the relevant sections of this dissertation that do so are outlined below. Since these questions overlap on issues somewhat, several of the sections will be relevant to more than one question.

- □ "Are the techniques for achieving fault tolerance and data security fully compatible? If not, what are the problems, and how can they be resolved? What tradeoffs are available?"

In Chapter 3, definitions were developed of what a fault and an error are in a secure computer system (Section 3.1). General computer architectures for fault-tolerant, secure computing were presented in order to deal with compatibility issues (Section 3.5). In several places in Chapter 3 and throughout the dissertation, fault tolerance techniques were shown to be effective against security problems. The extension of PFMs is an excellent example of a fault tolerance technique that solves both accidental and deliberate faults: the detection of computer virus errors, and of control flow errors due to transient and intermittent faults (Section 3.4.2). The application to fault-tolerant, secure computing of a basic design technique for MLS systems was discussed in Section 3.7.

The entirety of Chapter 4 is relevant to the above questions. It directly follows several of the leads from [Turn86] by delving into specific examples of fault tolerance mechanisms that cause security problems.

Lastly, in Chapter 5, details of how to extend a basic NVP system design for fault-tolerant, secure operation were presented (Sections 5.1).

In general, the above issues from [Turn86] are partly political and partly technical. The political aspect is related to the definition of compromise (see security glossary) and how it should be interpreted in an fault-tolerant, secure computing environment (see Section 3.1).

- "How does the architectural design for fault tolerance impact the design for security, and visa versa? How can the designs be made compatible?"

In Chapter 3, it is shown that the large size of the critical state that a secure system can contain makes it difficult to protect it from faults (Section 3.3). Furthermore, Section 3.5 is relevant to the above questions, and the partial definition of requirements for a class A2 secure system, as presented in

Section 3.8.1, includes the coping with faults.

Due to the overlap of issues addressed by these questions, all of Chapter 4 is again relevant to the above questions. Lastly, Section 6.4.2 outlines what extensive, future work is needed in order to make fault-tolerant and secure computer designs compatible.

□ "Can data security be gracefully degrading?"

Section 3.8 addresses this half of the degradation question in depth. The design choices presented in that section must be included in a design that can properly trigger security degradation recovery. This should be tied to, if not be a part of, the fault tolerance detection and recovery mechanisms. The second half of the degradation issue is discussed in Section 6.4.1 as future research work.

In summary, the questions in [Turn86] have led to the beginnings of design approaches for fault-tolerant, secure computers.

In dependable system design (computer security) a model of the faults (threats) to cope with, as well as a model of the computing environment, are essential in any research project or for any specific computer design. An <u>informal model of security threats</u> addressed in this dissertation is presented in Section 3.2.

This model is considered a research contribution, in itself, for the following reasons: (a) it is original, since it is in terms of a fault, error, and failure analysis of security threats, thus providing a new perspective, (b) the model provides a clear characterization of the threats, which facilitates the devising of countermeasures, (c) the model includes the representation of the inter-relationships between security threats, which is needed for construction of complete countermeasures (e.g., to prevent denial-of-service completely, computer viruses must also be prevented [see Figure 3.14]), and (d) such a model can aid in the development of formal models.

This model has proven its usefulness by motivating several new countermeasures to old threats (e.g., computer viruses, Trojan horses). It has also confirmed an observation about covert channels presented in [Dobs86] (i.e., that NVP can prevent programs from using some covert channels, see Section 3.4.6).

Partial solutions to service- and intruder-generated denial-of-service have been presented in detail (Sections 3.4.1 and 2.3.3 respectively). Additionally, denial of a function's (hardware and/or software) correct performance, due to deliberate design faults introduced by a designer, has been added to the categories of denial-of-service (termed insider-generated, see Section 3.2). Design fault tolerance has been shown to provide a solution to this integrity based form of denial-of-service (see Sections 2.3.1, 3.4.3, 3.4.6, 3.6, and Chapter 5).

The three solution approaches presented can be further characterized. First, all are directed towards prevention of denial-of-service. This was chosen since the potential damage of an actual denial is viewed as too great. For example, denial-of-service of a battlefield communication network could led to catastrophic results.

Second, all the schemes can be expensive in resources used to ensure guaranteed access. For a specific application to use these solutions, and perhaps prevention in general, it must be highly-critical for the investment to be cost-effective.

The scheme presented to deal with part of the service-generated denial-of-service case is not for general purpose systems. The interfaces to the computer system they provide to a user and application designer are very different from the current off-the-shelf computer systems.

Lastly, this research has explored possible methods for the specification of denial-of-service concerns. Methods that incorporate variables that hold numbers to represent time are preferred over representations of systems in temporal logic. Temporal logic represents the time of events by only describing their relative order. Denial-of-service requires real-time constraints as they are used in real-time systems (see Sections 3.2 and 3.4.1).

An enhanced definition of <u>malicious logic</u> has been derived that motivates the use of a fault tolerance perspective for devising countermeasures. The derivation of the basic properties of such an attack has been contributed (e.g., malicious logic tries to hide from detection, see the introduction to Chapter 2, Sections 2.1 and 2.2).

Several countermeasures to the forms of malicious logic presented have been devised and discussed. In fact, for the deliberate design fault type of malicious logic, an experiment was conducted to explore the natural resistiveness of one such countermeasure against <u>injected malicious logic</u>. It was observed that maliciously trying to force coincident, similar errors in the NVP-based airplane Autoland system [Aviz87b] was hard, and often not possible. However, forcing coincident, dissimilar errors was often easy to do (i.e., both observations are valid only in relation to the assumptions made for the experiment, see Chapter 5).

The connection between fault tolerance and computer security has been made by the observation, that the effects of malicious logic can be classified under the fault class of "by intent" [Aviz87a]. The use of design fault tolerance and NVP is alluded to in [Dobs86], however, the possibility of the deliberate nature of the faults is not mentioned. This dissertation has taken a first look at how design diversity—in both hardware and software—could be used against malicious logic.

6.2 APPLICABILITY OF THIS WORK

In Chapter 2, it was stated that our work was applicable to highly critical systems. The reasoning behind this was that only such systems could justify the expense of our protective measures. While this is still true, it has been observed that malicious logic can hit almost any type of system.[1]

Attacks on computer systems can originate from many sources. From a spy trying to obtain sensitive information, from a malicious person trying to be a nuisance to others, and from an inquisitive person, who means no harm, but just wants to see if an attack can be successful.

This wide spectrum of potential attackers increases the chances that almost any system will be targeted for penetration or subversion [Myer80]. Thus, the increase in attacks on many different types of systems may result in justification for the application of the protection mechanisms introduced here, or others, to all computer systems.

6.3 RELATION TO PREVIOUS WORK

The purpose of this section is to outline and compare work that is related to and precedes this dissertation. The references below are not exhaustive, but are extensive and representative of previous work. Most of these references,

[1]A computer virus has recently infected many privately owned personal computers. The virus has made its way via pirated and public domain software physically transmitted over computer networks or on shared diskettes.

except the more recent ones, discuss reliability rather than fault tolerance. This is not surprising, and was in fact expected.

For clarity, it should be clear, that fault tolerance is an approach to achieve system reliability, just as is fault avoidance. The proper system design approach employs a balanced mix of both.

In all the references below one common theme is present. It is that reliability (of software and hardware) is needed to ensure that security mechanisms work constantly; otherwise the system must be made to stop functioning [Shan77]. This is currently a well accepted principle of computer security.

The early work in [Molh73] was actually concerned with both accidental and deliberate hardware failures, and their impact on security. One threat described in detail are deliberate design faults inserted during maintenance, and thus, they are deliberate interaction faults. An example of this is a CPU fault that can be triggered by an unusual sequence of instructions. This proposed fault (trap door) would cause a system's protection mechanisms to be temporarily bypassed.

Solutions proposed in [Molh73] for all types of failures are: (a) periodic testing of hardware via software and/or microcode, (b) redundancy such as TMR and redundant control signal paths to prevent failures that can cause holes in security, and (c) fail-secure design.

Fail-secure design protects secure information from compromise (and destruction) regardless of failure. To achieve this, detection of failure conditions is essential. However, fail-soft designs are not by themselves fail-secure, since during degradation hardware vital to security may be lost.

The above work describes some of the difficulties involved in trying to detect deliberate hardware design faults with run-time mechanisms. No effective schemes are presented to deal with these faults. Additionally, the schemes presented to handle accidental faults are dated, since testing is ineffective in locating transient faults (i.e., transients disappear by the time the tests start, leaving behind errors).

[Fabr73] is discussed in Section 3.5, and involves the design of a system to maintain process separation for data privacy in the presence of faults.

A survey describing operating system structures to support countermeasures for a wide range of security threats appears in [Lind76]. The viewpoint taken, is that certain operating system facilities could be used to achieve both security and reliable software. This connection is made simply by pointing out that reliable software is essential to secure system design.

Faults and inadequate fault recovery mechanisms are potential sources of security violations. For example, unprotected checkpointed data can be read by unauthorized subjects (thus leading to compromise), or modified by unauthorized subjects (thus leading to penetration or denial-of-service). Solutions proposed for both reliable and secure software are: small protection domains and extended typed objects.

Small protection domains are essentially an implementation of the least privilege principle (see security glossary). A program or submodule of a program is only allowed to have access to resources that are essential to its job. Extended typed objects is very similar to type enforcement used in LOCK (see security glossary). The results of using both of these mechanisms are: (a) natural error containment boundaries, and (b) small system state needed for recovery.

In [Hsia79], it is proposed that the added mechanisms used to achieve reliability may be used, possibly with a few extensions, to detect some security violations. This is exactly the point of view taken in this dissertation and is demonstrated in Sections 2.3, 3.4, 3.6, and Chapter 5. However, the use of redundant copies of information (e.g., backup copies of files), is considered to increase a system's exposure to security violations.

[Namj82] is also discussed in Section 3.5, and it presents a design for protecting the access controls of a secure computer from physical faults and some accidental design faults.

[Kak83] is an example of an attempt to use one technique to provide both fault tolerance and computer security. A scheme for a joint encryption and error–correction code is proposed. Whereas this scheme does not yield a computationally practical approach, it does demonstrate a logical link between fault tolerance and computer security (also see [McEl81]). This idea of one set of mechanisms to solve both problems was first discussed in Chapter 1 and has been a major goal of this dissertation.

A rather limited view on the nature of reliability, security, and safety can be found in [Frie84]. In this paper, a supposedly fundamental distinction between both security and safety with reliability is made. Security and safety are properties of a system in which a designer defines what are the acceptable states for the computer system. However, reliability only has to do with transitions between states of a system that are related to physical properties of the system (i.e., such as failure rates of components). The problem with this distinction is that reliability, and more generally the dependability of a system, is also defined by the designer. A designer chooses or defines which fault classes will be handled by the computer design. This selection takes place for the obvious reason that addressing all faults is too expensive, if possible at all.

[Dobs86], [Neum86], and [Turn86] have all been frequently referred to throughout this dissertation. As such, only a few comments are appropriate here. One problem with the work in [Dobs86] is that it almost completely ignores the deliberate and insidious nature that faults can possess in a secure system. This property can make errors generated by such faults very difficult to detect, as well as make the faults themselves very hard to locate.

The work presented in [Neum86] proposes the current design technique used for MLS systems as a starting point for critical system design (i.e., fault-tolerant, secure, and safe). This is justified by the natural error containment boundaries that exist for different levels of sensitive data in an MLS system, and because of the rigorous design approach used to develop such systems. However, it completely ignores the impact that local and global error recovery can have on security and on the proposed standard design approach (see Section 3.7).

In [Koga82] [Desw86], intrusion tolerance rather than intrusion avoidance is used to foil penetrators in a local area network environment. Cryptography is an example of an intrusion tolerance technique, whereas access controls and identification are an example of intrusion avoidance. The main idea is to break up data into fragments and scatter them around the entire network (i.e., fragmentation-scattering). Thus, any one or several fragments are meaningless to a penetrator. Several fault tolerance techniques are integrated with fragmentation-scattering, such as redundant storage of each fragment in order to make the network tolerant of faults and intrusions.

The approach taken in [Koga82] [Desw86] is obviously complementary to the viewpoint taken in this dissertation. This can be seen by noting that both rely on the basic fault tolerance viewpoint, that avoidance of anything is impossible to guarantee, and that therefore run-time mechanisms are needed as a defense.

As previously mentioned in Section 2.2.1 and [Jose87], the approach used in [Glig85a] for coping with the denial-of-service threat clearly uses fault tolerance techniques. The setting of a timer on the acquisition of some service, and the expiration of that timer on the occurrence of denial-of-service is nothing more than a watchdog timer detecting a timing error. The switching to alternative services, which follow the same functional specification as the denied service, is nothing more than sparing redundancy. The work in this dissertation has also applied techniques from fault tolerance toward a solution to this threat.

Last, [Wu87] describes one particular feature of a secure computer that must be reliable. The storage of a master key that is used to encrypt other encryption keys is a part of the critical state of a secure system. To achieve reliability the master key is decomposed into several subkeys of which only a subset is needed to form the master key. The important parameters in a system design, are the number of subkeys that are essential and the time required for reconstruction of the master by key combination.

6.4 SUGGESTIONS FOR FUTURE WORK

6.4.1 AN UNANSWERED QUESTION FROM [TURN86]

In Section 3.8, we addressed the question: "Can data security be gracefully degrading?" from [Turn86]. However, this was only half of the problem. The other question not addressed from [Turn86] is: "Can gracefully degrading systems [i.e., systems degrade by switching between different levels of service] be data secure? Is there a difference in achieving each?"

This unanswered question is important for computer security, since degradation could cause: (a) covert channels—the way degradation is done may convey information (e.g., the sequence of subsystem shut downs and repairs can signal information), (b) denial-of-service—unnecessary degradation due to loss of integrity of control data (e.g., malicious modification of health and status messages to GOS, in the AOSP design, (see fault tolerance glossary) can lead to the turning off of healthy processors), and (c) compromise—due to incompatible policies (e.g., availability versus security: provide emergency network communication even upon loss of cryptographic facilities).

Such security failures as (a), (b) and (c) above could conceivably be caused by accidental faults causing degradation, and deliberate faults forcing degradation. In any case, the design of degradation sequences in a fault-tolerant, secure computer system[2] should use the fail-secure system design philosophy discussed in Section 6.3 and in [Molh73] (i.e., degradation should not result in compromise, [destruction of data] nor deliberate denial-of-service).

6.4.2 SECURITY POLICIES AND MODELS WITH FAULT TOLERANCE CONSIDERATIONS

In order to integrate fault tolerance and security into a truly fault-tolerant, secure (i.e., MLS) computer design, fault tolerance concerns and mechanisms used should be included in the system's security policy and model. The reasons for this are both political and technical.

Politically, the security community does not seem to support (new) design approaches that are not represented in a form that they are accustomed to. Technically, as discussed in Chapter 4 and Section 6.4.1, fault tolerance can

[2]For any computer system to be able to degrade in performance, and/or functionality, it must contain fault tolerance mechanisms to detect when to degrade, and to control the actual degradation.

have a definite impact on the security of a system. Additionally, a policy and formal model provide credibility to a design, and furthermore, the rigor required to construct a model may uncover additional problems and/or similarities between fault tolerance and security.

Thus, the suggestion to derive policies and models with fault tolerance concerns built-in, is viewed here as an essential step toward an actual design of a fault-tolerant, secure computer. Below, is a suggested outline of the issues that should be addressed in the construction of such policies and models.

To derive an adequate policy, the following issues should be addressed: (a) is a fail-secure system design required? (b) which architectural option (i.e., option A-fault masking, B-fail-stop, and C-allow a security relevant error), as presented in Section 3.5, is desired, and in what environments should it be used? (c) for highly available systems, when and where should degradable security be allowed and under what restrictions (e.g., no covert channels)?, and (d) when and where should extensions, similar to those defined in Section 3.8.1 for class A2 systems be used? For example, for item "c" above, if option-C is chosen, then a new definition of compromise would have to be derived and incorporated into the security model (see Section 3.1 and [Turn86]).

To derive an appropriate (formal) model of a fault-tolerant, secure computer system the following issues should be considered: (a) how should a fail-secure system design be represented in standard security models? (b) how can degradable security (see Section 3.8) be represented and what are the solutions to the problems mentioned in Section 6.4.1?, and (c) how can basic fault tolerance mechanisms, such as error detection and recovery, be represented (e.g., as subjects)? An additional problem with representing fault tolerance mechanisms, is that they are only guaranteed to work in relation to a specific set of faults and to a derived probability (i.e., coverage). This seems to impact the way a fail-secure design would be (formally) modeled.

6.4.3 SECURITY POLICIES, MODELS, AND MECHANISMS TO ADDRESS DENIAL-OF-SERVICE

To derive a complete solution to the denial-of-service threat a detailed analysis of the problem is suggested. This analysis should use a framework composed of techniques from both computer security and fault tolerance. This would combine the formal analysis used in many secure system designs [Cerf85], with the case-by-case analysis of specific faults and errors used in fault tolerance design. The two approaches fit quite well together, and are described below.

First, it is necessary to clearly define the policy or policies (i.e., a policy may be required for different environments, such as a single computer or a network) addressing the denial-of-service threat by determining: (a) a standard and accepted definition of denial-of-service in a specific environment, (b) where and when to worry about denial-of-service, and (c) what consists of a basic set of defenses [DoD87].

Next, in order to derive a formal model(s) of denial-of-service, and mechanisms to cope with it (i.e., again, a separate model and mechanism may be needed for each environment), a through understanding of the threat is likely to be necessary. Therefore, the steps below are recommended to acquire and document the necessary knowledge base.

(1) Perform a careful and detailed classification of all known cases of denial-of-service into classes (e.g., service-generated [Glig83]). Document all known cases in detail.

(2) Determine all interactions of denial-of-service with all other security threats and anomalies (see Figures 3.2, 3.3, 3.5 to 3.12, 3.14, and 3.15a).

(3) Perform experiments (in different environments) on the ease of causing a denial-of-service, on the varying effects of different instances of denial-of-service, and to derive new forms of denial-of-service.

(4) Derive solutions for each specific case of denial-of-service documented by task 1 through 3 above.

(5) Search for commonalities of the types and effects of the denial-of-service cases documented, and of the specific solutions derived.

(6) The five previous steps, or tasks, were essentially meant to form an extensive informal model of the denial-of-service threat in multiple environments. Now, with this information attempts can proceed to derive one or more formal models of denial-of-service.

Whereas some of the above tasks have been attempted in the past [Glig83] [Glig85a], these attempts have been completely informal, incomplete (e.g., this dissertation has defined a third class of denial-of-service, called insider-generated, see Section 3.2), and have not used an integrated fault tolerance and computer security approach as suggested above.

6.4.4 REMAINING ISSUES IN FAULT-TOLERANT, SECURE COMPUTING

Several basic issues still remain to be explored in fault-tolerant, secure computing. First and foremost is the problem of secure global recovery in MLS systems (this was discussed in Chapter 4 and in [Turn86]). Some effort has been spent on this issue, but still more work is needed to derive an adequate approach.

The approach for detecting computer virus errors (see Section 3.4.2) includes the overhead of encryption. Further work is needed to devise a similar scheme, based on [Schu87], that does not utilize encryption. This would possibly result in a detection mechanism that was only resistant to viral infections, not completely virus proof.

The bandwidths (or range of bandwidths) for the four covert channels presented in Chapter 4 need to be calculated. Currently, only a qualitative estimate has been provided. In order to determine quantitative bandwidths, each channel should be analyzed with a specific computer design in mind. Additionally, these four channels were meant only as examples of possible covert channels. More work is needed to find other general channels (i.e, only due to general fault tolerance mechanisms outside of a specific computer design), as well as specific cases in existing fault-tolerant computers.

Are other fault tolerance techniques and mechanisms applicable to security problems? While this dissertation has been a good first step, the next section will suggest an approach to continue the perspective used here.

Lastly, examples of malicious logic in hardware are needed [Faul82]. These are needed to provide a clear understanding of this threat, and will be used to ensure proper countermeasures (e.g., hardware design diversity, see Section 3.6). These examples should have the same properties as all forms of malicious logic. Several such properties are: hidden in complexity, hard to find by testing and mechanical inspection, triggerable, and can be mistaken for accidental design faults.

6.4.5 EXTENSION OF FAULT, ERROR, AND FAILURE CLASSIFICATIONS

The fault, error, and failure classification scheme presented in [Aviz87a] was used in Section 3.2 (see Tables 3.1 and 3.2) to capture characteristics of deliberate faults, errors, and failures. The design of this classification scheme was influenced by observed characteristics of naturally occurring, and human-made accidental faults. It is used as the beginning step in a design paradigm for fault-tolerant computer design.

Using this classification scheme has been beneficial in this dissertation for the construction of an informal, fault tolerance oriented model of security threats. It was essential to determine what security threats could be viewed as faults or errors in order to attempt to find fault tolerance countermeasures (i.e., detection, recovery, or masking approaches). For example, the determination that a computer virus is both a fault and an error (see Figure 3.1) directly led to the idea of applying Program Flow Monitors to detect computer virus errors, and N-Version Programming to prevent computer virus faults from generating computer virus errors (see Sections 3.4.2 and 3.6).

The basic classification scheme [Aviz87a] can be used to specify particular instances of a security threat (e.g., a non-evolving computer virus). This can then be used to pick an appropriate detection mechanism for the one (specific) case specified. However, deliberate attacks can take many forms, so that a more encompassing description is also desirable.

It is proposed that the classification scheme presented in [Aviz87a] be extended to include other characteristics of deliberate faults and their consequent errors (e.g., how vulnerable to exposure does a malicious engineer make himself by implanting different types of malicious logic). Once this is done, the classification attempt represented by Tables 3.1 and 3.2 should be redone. It is envisioned that this work will lead to the additional insight necessary to determine whether other fault tolerance techniques are applicable to security problems.

Two important points must be made clear to the reader before the close of this section. First, we propose an extension rather than a redesign of the classification scheme. This is due to two simple facts: (1) deliberate faults can easily force errors that appear to have been caused by accidental faults, and (2) deliberate faults can be made to appear as if they occurred accidentally (e.g., deliberate software design faults).

Lastly, the existing classification scheme already includes the intent characteristic of faults. Thus, we have already informally modeled the deliberate nature (or malice) of the security threats addressed in this dissertation.

6.4.6 DATABASE SECURITY

This topic has not been directly addressed in this dissertation. However, it is obviously important, since most computer systems have some form of database. Points of interest, that would guide future work in applying the perspective used in this dissertation to secure databases are listed below.

(1) Address threats unique to databases (e.g., inference in statistical databases [Denn82]), by determining how they fit into the informal models presented in this dissertation, and by determining what fault tolerance techniques, if any, are applicable.

(2) Determine what threats, already addressed, are present in secure databases.

(3) Study new instances of threats already addressed (see footnotes 1 and 2 in Chapter 3), and perform a similar analysis as outlined in item 1 above.

REFERENCES AND BIBLIOGRAPHY

[Aero86] The Aerospace Corp., "Prevention and Detection of Trojan Horses," draft of internal document, June 1986.

[Abbo87] R.J.Abbott, "Resourceful Systems," The Aerospace Corp., draft of internal document, August 1987.

[Air84] U.S. Airforce, Automatic Data Processing (ADP) Security Policy Procedures, and Responsibilities, AF Regulation 205-16, Headquaters, Washington, D.C., August 1984.

[Avia87] Aviation Week & Space Technology, "Airbus 320, The New Generation Aircraft," Feb. 2, 1987, pp.45-66.

[Aviz84] A.Avizienis, and J.P.Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," Computer, Vol. 17, No. 8, August 1984, pp.67-80.

[Aviz85a] A.Avizienis, "The N-Version Approach to Fault-Tolerant Software," IEEE Trans. on Soft. Eng., Vol. SE-11, No. 12, Dec. 1985, pp.1491- 1501.

[Aviz85b] A.Avizienis et al., "The UCLA DEDIX System: A Distributed Testbed for Multiple - Version Software," 15th Int'l Symp. on Fault-Tolerant Computing Systems, June 1985, pp.126-134.

[Aviz85c] A.Avizienis, "Fault-Tolerant Computing Systems," UCLA Class Notes, Computer Science Department, Jan. 1985.

[Aviz86] A.Avizienis, and J-C.Laprie, "Dependable Computing: From Concepts to Design Diversity," Proc. of the IEEE, Vol. 74, No. 5, May 1986, pp.629-638.

[Aviz87a] A.Avizienis, "A Design Paradigm for Fault-Tolerant Systems," AIAA Computers in Aerospace VI Conf., Oct. 1987, pp.52-57.

[Aviz87b] A.Avizienis, M.R.Lyu, and W.Schuetz, "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software," UCLA Computer Science Dept. Tech. Report CSD-870060, Nov. 1987. To appear in the 18th Int'l Symp. on Fault-Tolerant Computing Systems, June 1988.

[Bamf86] J.Bamford, "The Walker Espionage Case," Proc. Naval Review, pp.111-119.

[Barb86] M.R.Barbacci, and J.M.Wing, "Specifying Functional and Timing Behavior for Real-Time Applications," CMU Software Engineering Institute Technical Report, ESD-TR-86-208, CMU/SEI-86-TR-4, Dec. 1986.

[Bard87] P.H.Bardell, W.H.McAnney, and J.Savir, Built-In Test for VLSI: Pseudorandom Techniques, John Wiley & Sons, New York, 1987.

[Bast85] F.B.Bastani, and I.L.Yen, "Analysis of an Inherently Fault Tolerant Program," Proc. of Compsac 85, IEEE Computer Society, Oct. 1985, pp.428-436.

[Bell75] D.E.Bell, and L.J.LaPadula, "Computer Security Model: Unified Exposition and Multics Interpretation," Tech. report ESD-TR-75-306, ADAO23588, The Mitre Corp., Bedford, Mass., June 1975.

[Bevi87] W.R.Bevier, W.A.Hunt, Jr., and W.D.Young, "Toward Verified Execution Environments," IEEE Symp. on Security and Privacy, April 1987, pp.106-115.

[Biba77] K.J.Biba, "Integrity Considerations for Secure Computer Systems," Mitre Technical Report TR-3153, Mitre Corp., Bedford, MA., April 1977.

[Boeb85a] W.E.Boebert, and R.Y.Kain, "A Practical Alternative to Hierarchical Integrity Policies," 8th National Computer Security Conf., Sept.30-Oct.3, 1985, pp.18-27.

[Boeb85b] W.E.Boebert, W.D.Young, R.Y.Kain, and S.A.Hansohn," Secure Ada Target: Issues, System Design, and Verification," IEEE Symp. on Security and Privacy, April 1985, pp.176-183.

[Boeb85c] W.E.Boebert, R.Y.Kain, and W.D.Young, "Secure Computing: The Secure Ada Target Approach," Scientific Honeyweller, Vol. 6, No. 2, July 1985, pp.1-17.

[Boor87] S.A.Boorman, and P.R.Levitt, "A New Battlefield: Software Warfare - Rising Form of Computer Sabotage May be Next Great Military Equalizer," Seattle Times, Sept. 20, 1987.

[Bour71] W.G.Bouricius, W.C.Carter, D.C.Jessep, P.R.Schneider, and A.B.Wadia, "Reliability Modeling for Fault-Tolerant Computers," IEEE Trans. on Computers, Vol. C-20, No. 11. Nov. 1971, pp.1306-1311.

[Brow73] P.S.Browne, "Taxonomy of Security and Integrity," Security and Privacy in Computer Security, L.J.Hoffman editor, Melville Publishing Co., Los Angeles, California, 1973, pp.369-378.

[Cart68] W.C.Carter, and P.R.Schneider, "Design of Dynamically Checked Computers," Proc. IFIP Congress, Vol. 2, Edinburgh, 1968, pp.878-888.

[Cart85] W.C.Carter, "Hardware Fault Tolerance," Chapter 2 of Resilient Computer Systems, Volume 1, editor T.Anderson, John Wiley & Sons, New York, 1985.

[Cerf74] V.G.Cerf, and R.E.Kahn, "A Protocol for Packet Network Interconnection," IEEE Trans. on Communications, Vol. COM-22, No. 5, May 1974, pp.637-648.

[Cerf85] V.G.Cerf, "Report of the Denial of Service Group," Proc. DoD Computer Security Center Invitational Workshop on Network Security, March 1985, pp.9-3 — 9-31.

[Cheh83] M.H.Cheheyl, M.Gasser, G.A.Huff, and J.K.Millen, "Verifying Security," ACM Computing Surveys, Vol. 13, No. 3, Sept. 1983, pp.279-339.

[Chen78] L.Chen, "Improving Software Reliability By N-Version Programming," Ph.D. Dissertation, UCLA Computer Science Department, Los Angeles, California, Eng-7843, August 1978.

[Clar87] D.D.Clark, and D.R.Wilson, "A Comparison of Commercial and Military Computer Security Policies," IEEE Symp. on Security and Privacy, April 1987, pp.184-194.

[Clyd87] A.R.Clyde, "Insider Threat Identification Systems," 10th National Computer Security Conf., Sept. 1987, pp.343-356.

[Cohe84] F.Cohen, "Computer Viruses: Theory and Experiments," 7th National Computer Security Conf., Sept. 1984, pp.240-263.

[Cohe85] F.Cohen, "Computer Viruses," Ph.D. Dissertation, Electrical Engineering Dept., University of Southern California, Los Angeles, California, Oct. 1985.

[Cohe87] F.Cohen, "A Cryptographic Checksum for Integrity Protection," Computers & Security, Vol. 6, No. 6, North-Holland, Dec. 1987, pp.505-510.

[Denn82] D.E.Denning, Cryptography and Data Security, Addison-Wesley, Reading, Massachusetts, 1982.

[Denn86] D.E.Denning, "An Intrusion-Detection Model," IEEE Symp. on Security and Privacy, April 1986, pp.118-131.

[Desw86] Y.Deswarte, J-C.Fabre, J-C.Laprie, and D.Powell, "A Saturation Network to Tolerate Faults and Intrusions," IEEE 5th Symp. on Reliability in Distributed Software and Database Systems, Jan. 1986, pp.74-81.

[Dobs86] J.E.Dobson, and B.Randell, "Building Reliable Secure Computing Systems out of Unreliable Insecure Components," IEEE Symp. on Security and Privacy, April 1986, pp.187-193.

[DoD83a] Transmission Control Protocol, MIL-STD-1778, August 1983.

[DoD83b] Internet Protocol, MIL-STD-1777, August 1983.

[DoD85a] Department of Defense Trusted Computer System Evaluation Criteria, DoD 5200.28-STD, Dec. 1985.

[DoD85b] Computer Security Requirements: Guidance for Applying the Department of Defense Trusted Computer System Evaluation Criteria in Specific Environments, CSC-STD-003-85, June 1985.

[DoD85c] Technical Rationale Behind "CSC-STD-003-85," CSC-STD-004-85, June 1985.

[DoD87] Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria, NCSC-TG-005, Version-1, July 1987.

[Down85] D.D.Downs, J.R.Rub, K.C.Kung, and C.S.Jordan, "Issues in Discretionary Access Control," IEEE Symp. on Security and Privacy, April 1985, pp.208-218.

[Ezhi86] P.D.Ezhilchelvan, and S.K.Shrivastava, "A Characterisation of Faults in Systems," IEEE 5th Symp. on Reliability in Distributed Software and Database Systems, Jan. 1986, pp.215-222.

[Fabr73] R.S.Fabry, "Dynamic Verification of Operating System Decisions," Comm. of the ACM, Vol. 16, No. 11, Nov. 1973, pp.659-668.

[Faul82] T.L.Faulkner, C.W.Bartlett, and M.Small, "Hardware Logic Design Faults - A Classification and Some Measurements," 12th Int'l Symp. on Fault-Tolerant Computing Systems, June 1982, pp.377-380.

[Fell87] J.Fellows, J.Hemenway, N.Kelem, and S.Romero, "The Architecture of a Distributed Trusted Computing Base," 10th National Computer Security Conf., Sept. 1987, pp.68-77.

[Frag85] J.Fraga, and D.Powell, "A Fault and Intrusion-Tolerant File System," 3rd IFIP Int'l Congress on Computer Security, Dublin, Ireland, August 1985, pp.203-218.

[Fran86] N.Francez, Fairness, Springer-Verlag, New York, 1986.

[FRBS87] Federal Reserve Bank of San Francisco, Research Department, "Controlling Payments System Risk," FRBSF Weekly Letter, August 14, 1987.

[Frei82] R.Freiburghouse, "Making Processing Fail-Safe," Mini-Micro Systems, May 1982, pp.255-264.

[Frie84] A.W.Friend, "What is the Difference Between Safety, Security, and Reliability?", Proc. IEEE 1987 Compcon, pp.289-293.

[Funa78] S.Funatsu et al., "Designing Digitial Circuits with Easily Testable Consideration," Proc. Int'l Test Conf., 1978, pp.98–102.

[Gall77] R.G.Gallager, "A Minimum Delay Routing Algorithm Using Distributed Computation," IEEE Trans. on Communications, Vol. COM–25, No. 1, Jan. 1977, pp.73–84.

[Gass88] M.Gasser, Building A Secure Computer System, Van Nostrand Reinhold, New York, 1988.

[GC84] Gemini Computers, System Overview: Gemini Trusted Multiple Microcomputer Base, Version 0, Carmel, California, 1984.

[Gerl82] M.Gerla, and L.Kleinrock, "Flow Control Protocols," Chapter 13 in Computer Network Architectures and Protocols, Editor: P.E.Green, Jr., Plenum Press, New York and London, 1982, pp.361–412.

[Gill87] G.C.Gilley, "Architectural Approaches to Transient Fault Protection," AIAA Computers in Aerospace VI Conf., Oct. 1987, pp.78–82.

[Glig83] V.D.Gligor, "A Note on the Denial–of–Service Problem," IEEE Symp. on Security and Privacy, April 1986, pp.139–149.

[Glig85a] V.D.Gligor, "Denial–of–Service Implications for Computer Networks," Proc. DoD Computer Security Center Invitational Workshop on Network Security, March 1985, pp.9–33 — 9–48.

[Glig85b] V.D.Gligor, "Analysis of the Hardware Verification of the Honeywell SCOMP," IEEE Symp. on Security and Privacy, April 1985, pp.32–43.

[Gogu82] J.A.Goguen, and J.Meseguer, "Security Policies and Security Models," IEEE Symp. on Security and Privacy, April 1982, pp.11–20.

[Gold80] J.Goldberg, "SIFT: A Provable Fault–Tolerant Computer for Aircraft Flight Control," Information Processing 80, S.H.Lavington, editor, North–Holland, New York, 1980.

[Good84] D.I.Good, B.L.Divito, and M.K.Smith, "Using the Gypsy Methodology," Tech. report, Institute for Computing Science, The University of Texas at Austin, June 1984.

[Good86] D.I.Good, R.L.Akers, and L.M.Smith, "Report on Gypsy 2.05," Tech. report ICSCA–CMP–48, Institute for Computing Science and Computing Applications, The University of Texas at Austin, Feb. 1986.

[Grau84] R.Graubart, "The Integrity–Lock Approach to Secure Database Management," IEEE Symp. on Security and Privacy, April 1984, pp.62–74.

[Gray86] J.Gray, "Why do Computers Stop and What Can be Done About it?," IEEE 5th Symp. on Reliability in Distributed Software and Database Systems, Jan. 1986, pp.3–12.

[Hami87] M.J.Hamilton (The Aerospace Corp.), and J.Crawford, System Software Support for Real-time Simulation Programming for Avionics, personal communication, NASA Langley Research Center, August 1987.

[Henn86] R.R.Henning, and S.A.Walker, "Computer Architecture and Database Security," 9th National Computer Security Conf., Sept. 1986, pp.216–230.

[Hers84] I.S.Herschberg, and R.Paans, "The Programmer's Threat: Cases and Causes," Computers & Security, Vol. 3, No. 4, North-Holland, New York, Nov. 1984, pp.263–272.

[Hsia79] D.K.Hsiao, D.S.Kerr, and S.E.Madnick, Computer Security, Academic Press, New York, 1979.

[Hua86] K.A.Hua, and J.A.Abraham, "Design of Systems with Concurrent Error Detection Using Software Redundancy," ACM/IEEE Fall Joint Computer Conf., Dallas, TX., Nov. 1986, pp.826–835.

[IEEE87] IEEE, Glossary, Computer Society Press Tutorial: Computer And Network Security, 1987, pp.413–424.

[Inte83] Intel, iAPX 286 Programmer's Reference Manual, Santa Clara, California, 1983.

[Jaha86] F.Jahanian, and A.K.Mok, "Safety Analysis of Timing Properties in Real-Time Systems," IEEE Trans. on Soft. Eng., Vol. SE-12, No. 9, Sept. 1986, pp.890–904.

[Jaha87] F.Jahanian, and A.K.Mok, "A Graph-Theoretic Approach for Timing Analysis and its Implementation," IEEE Trans. on Computers, Vol. C-36, No. 8, August 1987, pp.961–975.

[Jose87] M.K.Joseph, "Towards the Elimination of the Effects of Malicious Logic: Fault Tolerance Approaches," 10th National Computer Security Conf., Sept. 1987, pp.238–244.

[Kak83] S.C.Kak, "Joint Encryption and Error-Correction Coding," IEEE Symp. on Security and Privacy, April 1983, pp.55–60.

[Kell86] J.P.J.Kelly, A.Avizienis, B.T.Ulery, B.J.Swain, R.T.Lyu, A.T.Tai, and K.S.Tso, "Multi-Version Software Development," Proc. IFAC Workshop SAFECOMP'86, Sarlat, France, Oct. 1986, pp.43–49.

[Kemm83] R.A.Kemmerer, "Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels," ACM Trans. on Computer Systems, Vol. 1, No. 3, August 1983, pp.256–277.

[Kemm85] R.A.Kemmerer, "Testing Formal Specifications to Detect Design Errors," IEEE Trans. on Soft. Eng., Vol. SE-11, No. 1, Jan. 1985, pp.32-43.

[Knig86] J.C.Knight, and N.G.Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," IEEE Trans. on Soft. Eng., Vol. SE-12, No. 1, Jan. 1986, pp.96-109.

[Koga82] Y.Koga, E.Fukushima, and K.Yoshihara, "Error Recoverable and Securable Data Communication of Computer Network," 12th Int'l Symp. on Fault-Tolerant Computing Systems, June 1982, pp.183-186.

[Kope85] K.Kopetz, and W.Merker, "The Architecture of Mars," 15th Int'l Symp. on Fault-Tolerant Computing Systems, June 1985, pp.274-279.

[Kueh69] R.E.Kuehn, "Computer Redundancy: Design, Performance, and Future," IEEE Trans. on Reliability, Vol. R-18, No. 1, Feb. 1969, pp.3-11.

[Lai88] N.Lai, and T.E.Gray, "Strengthening Discretionary Access Controls to Inhibit Trojan Horses and Computer Viruses," 1988 USENIX Technical Conf., June 1988.

[Lamp73] B.W.Lampson, "A Note on the Confinement Problem," Comm. of the ACM, Vol. 16, No. 10, Oct. 1973, pp.613-615.

[Land85] B.Landreth, Out of the Inner Circle, Microsoft Press, Bellevue, Washington, 1985.

[Landw81] C.E.Landwehr, "Formal Models for Computer Security," ACM Computing Surveys, Vol. 13, No. 3, Sept. 1981, pp.247-278.

[Landw84] C.E.Landwehr, and J.M.Carroll, "Hardware Requirements for Secure Computer Systems: A Framework," IEEE Symp. on Security and Privacy, April 1984, pp.34-40.

[Leve85] N.G.Leveson, "Software Safety," Chapter 7 of Resilient Computing Systems, Volume 1, editor T.Anderson, John Wiley & Sons, New York, 1985.

[Leve86] N.G.Leveson, "Software Safety: Why, What, and How," ACM Computing Surveys, Vol. 18, No. 2, June 1986, pp.125-163.

[Lind76] T.A.Linden, "Operating System Structures to Support Security and Reliable Software," ACM Computing Surveys, Vol. 8, No. 4, Dec. 1976, pp.409-445.

[Lipn75] S.B.Lipner, "A Comment on the Confinement Problem," Operating Systems Review, Vol. 9, No. 5, Nov. 1975, pp.192-196.

[Loep85] K.Loepere, "Resolving Covert Channels within a B2 Class Secure System," Operating Systems Review, July 1985, pp.9–28.

[Lyu88] M.R.Lyu, "A Design Paradigm for Multi-Version Software," Ph.D. Dissertation, UCLA Computer Science Department, Los Angeles, California, May 1988.

[Mahm88] A.Mahmood, and E.J.McCluskey, "Concurrent Error Detection Using Watchdog Processors — A Survey," IEEE Trans. on Computers, Vol. C–37, No. 2, Feb. 1988, pp.160–174.

[McEl81] R.J.McEliece, and D.V.Sarwate, "On Sharing Secrets and Reed-Solomon Codes," Comm. of the ACM, Vol. 24, No. 9, Sept. 1981, pp.583–584.

[Mell82] P.M.Melliar-Smith, and R.L.Schwartz, "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System," IEEE Trans. on Computers, Vol. C–31, No. 7, July 1982, pp.616–630.

[Molh73] L.M.Molho, "Hardware Aspects of Secure Computing," Security and Privacy in Computer Security, L.J.Hoffman editor, Melville Publishing Co., Los Angeles, California, 1973, pp.351–365.

[Myer80] P.A.Myers, "Subversion: The Neglected Aspect of Computer Security," Masters Thesis, Naval Postgraduate School, Monterey, California, 1980.

[Namj82] M.Namjoo, and E.J.McCluskey, "Watchdog Processors and Capability Checking," 12th Int'l Symp. on Fault-Tolerant Computing Systems, June 1982, pp.245–248.

[Neum78] P.G.Neumann, "Computer Security Evaluation," AFIPS Conf. Proceedings, Vol. 47, NCC, 1978, pp.1087–1095. Reprinted: Advances in Computer Security, editor R.Turn, Artech House, 1981, pp.41–49.

[Neum86] P.G.Neumann, "On Hierarchical Design of Computer Systems for Critical Applications," IEEE Trans. on Soft. Eng., Vol. SE–12, No. 9, Sept. 1986, pp.905–920.

[Osde79] S.Osder, "The DC–9–80 Digital Flight Guidance System's Monitoring Techniques," AIAA Guidance and Ccntrol Conf., August 1979, pp.64–79.

[Paan83] R.Paans, and G.Bonnes, "Surreptitious Security Violation in the MVS Operating System," Computers & Security, Vol. 2, No. 2, North-Holland, June 1983, pp.144–152.

[Pete72] W.W.Peterson, and E.J.Weldon,Jr., Error-Correcting Codes, The MIT Press, Cambridge, Massachusetts, 1972.

[Port85] S.Porter, and T.S.Arnold, "On the Integrity Problem," 8th National Computer Security Conf., Sept.30—Oct.3, 1985, pp.15–17.

[Pozz86] M.M.Pozzo, and T.E.Gray, "A Model for the Containment of Computer Viruses," AIAA/ASIS/DODCI 2nd Aerospace Computer Security Conf., Dec. 1986, pp.11–18.

[Prad86] D.K.Pradham, editor, Fault-Tolerant Computing: Theory and Techniques, Volumes 1&2, Prentice-Hall, New Jersey, 1986.

[RADC85] Raytheon Company, "Advanced Onboard Signal Processor (AOSP) Phase IIA Development," RADC-TR-85-36, Vol. I, Feb. 1985.

[RADC87] R.Turn, "System Security and its Relationship to Fault Tolerance," Application of Fault Tolerance Technology, Vol. 4, RADC Tech. Report, Working Document, SDIO BM/CCC Processor and Algorithm Working Group, Oct. 1987.

[Ragh85] C.S.Raghavendra, M.Gerla, and A.Avizienis, "Reliable Loop Topologies for Large Local Computer Networks," IEEE Trans. on Computers, Vol. C-34, pp.46–55.

[Rama81] C.V.Ramamoorthy, and F.B.Bastani, "Software Reliability: Status and Perspectives," IEEE Trans. on Soft. Eng., July 1981, pp.354–371.

[Rand75] B.Randell, "System Structure for Fault Tolerance," IEEE Trans. on Soft. Eng., Vol. SE-1, No. 2, March 1975, pp.220–232.

[Renn78] D.A.Rennels, "Architectures for Fault-Tolerant Spacecraft Computers," Proc. of the IEEE, Vol. 66, No. 10, Oct. 1978, pp.1255–1268.

[Renn84] D.A.Rennels, "Fault-Tolerant Computing — Concepts and Examples," IEEE Trans. on Computers, Vol. C-33, No. 12, Dec. 1984, pp.1116–1129.

[Renn86] D.A.Rennels, and S.Chau, "A Self-Exercising Self-Checking Memory Design," 16th Int'l Symp. on Fault-Tolerant Computing Systems, July 1986, pp.358–363.

[Rose81] E.C.Rosen, "Vulnerabilities of Network Control Protocols: An Example," ACM Sigsoft, Soft. Eng. Notes, Vol. 6, No. 1, Jan. 1981, pp.6–8.

[Rouq86] J.C.Rouquet, and P.J.Traverse, "Safe and Reliable Computing on Board the Airbus and ATR Aircraft," Proc. IFAC Workshop SAFECOMP'86, Sarlat, France, Oct. 1986, pp.93–97.

[Sayd87] O.S.Saydjari, J.M.Beckman, and J.R.Leaman, "Locking Computers Securely," 10th National Computer Security Conf., Sept. 1987, pp.129–141.

[Scha77] M.Schaefer, B.Gold, R.Linde, and J.Scheid, "Program Confinement in KVM/370," Proc. ACM National Conf., Oct. 1977, pp.404–410.

[Sche86] R.R.Schell, and D.E.Denning, "Integrity in Trusted Database Systems," 9th National Computer Security Conf., Sept. 1986, pp. 30–36.

[Schu87] M.A.Schuette, and J.P.Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," IEEE Trans. on Computers, Vol. C–36, No. 3, March 1987, pp.264–276.

[Schue87] W.Schuetz, "Diversity in N–Version Software: An Analysis of Six Programs," Master Thesis, UCLA Computer Science Department, Los Angeles, California, Dec. 1987.

[Shan77] K.S.Shankar, "The Total Computer Security Problem," IEEE Computer, June 1977, pp.50–73. Reprinted: Advances in Computer System Security, editor R.Turn, Artech House, 1981, pp.25–40.

[Shen86] J.P.Shen, "Behavior Based Fault Tolerance," Research Proposal to the Office of Naval Research, CMU, Pittsburg, PA., 1986.

[Shir81] L.J.Shirley, and R.R.Schell, "Mechanism Sufficiency Validation by Assignment," IEEE Symp. on Security and Privacy, April 1981, pp.26–32.

[Smit86] T.A.Smith, "User Definable Domains as a Mechanism for Implementing the Least Privilege Principle," 9th National Computer Security Conf., Sept. 1986, pp.143–148.

[Stol79] L.A.Stolte, and N.C.Berglund, "Design for Testability of the IBM System/38," Proc. Int'l Test Conf., 1979, pp.29–36.

[Tane81] A.S.Tanenbaum, Computer Networks, Prentice–Hall, Englewood Cliffs, New Jersey, 1981.

[Tayl80] D.J.Taylor, D.E.Morgan, and J.P.Black, "Redundancy in Data Structures: Improving Software Fault Tolerance," IEEE Trans. on Soft. Eng., Vol. SE–6, No. 6, Nov. 1980, pp.585–594.

[Thom84] K.Thompson, "Reflections on Trusting Trust," Comm. of the ACM, Vol. 27, No. 8, August 1984, pp.761–763.

[Toma85] S.P.Tomas, and J.P.Shen, "A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems," Proc. IEEE Int. Conf. Computer Design: VLSI Computers, Port Chester, N.Y., Oct. 1985, pp.531–539.

[Tso87] K.S.Tso, and A.Avizienis, "Community Error Recovery in N–Version Software: A Design Study With Experimentation," 17th Int'l Symp. on Fault–Tolerant Computing Systems, July 1987, pp.127–133.

[Turn86] R.Turn, and J.Habibi, "On the Interactions of Security and Fault Tolerance," 9th National Computer Security Conf., Sept. 1986, pp.138–142.

[Voyd83] V.L.Voydock, and S.T.Kent, "Security Mechanisms in High-Level Network Protocols," ACM Computing Surveys, Vol. 15, No. 2, June 1983, pp.135–171.

[Voyd85a] V.L.Voydock, and S.T.Kent, "Security in High-Level Network Protocols," IEEE Communications Magazine, Vol. 23, No. 7, July 1985, pp.12–24.

[Voyd85b] V.L.Voydock, and S.T.Kent, "Security Mechanisms in a Transport Layer Protocol," Computers & Security, Vol. 4, No. 4, North-Holland, New York, Dec. 1985, pp.325–341.

[Wall84] J.J.Wallace, and W.W.Barnes, "Designing for Ultrahigh Availability: The Unix RTR Operating System," IEEE Computer, Vol. 17, No. 8, August 1984, pp.31–39.

[Wens78] J.H.Wensley et al., "SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," Proc. of the IEEE, Vol. 66, No. 10, Oct. 1978, pp.1240–1255.

[Will73] M.J.Y.Williams, and J.B.Angell, "Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Additional Logic," IEEE Trans. on Computers, Vol. C–22, No. 1, Jan. 1973, pp.46–60.

[Wing86] J.M.Wing, and M.R.Nixon, "Extending Ina Jo with Temporal Logic," IEEE Symp. on Security and Privacy, April 1986, pp.2–13.

[WIPC87] Draft Report of the Invitational Workshop on Integrity Policy in Computer Information Systems (WIPCIS), Bentley College, Massachusetts, Oct. 1987.

[WSJ87] The Wall Street Journal, "Costly Bugs," Jan. 28, 1987, p.17.

[Wu87] T.Wu, and K.Chen, "Reliability and Key-Protection for Computer-Security Systems," IEEE Trans. on Reliability, Vol. R–36, No. 1, April 1987, pp.113–116.

[Youn87] W.D.Young, and J.McHugh, "Coding for a Believable Specification to Implementation Mapping," IEEE Symp. on Security and Privacy, April 1987, pp.140–148.

[Yu87] C.F.Yu, "A Formal Specification and Verification Method for Denial of Service in Computer Systems," Ph.D. Dissertation, University of Maryland, August 1987.

[Yu88] C.F.Yu, and V.D.Gligor, "A Formal Specification and Verification Method for the Prevention of Denial of Service," IEEE Symp. on Security and Privacy, April 1988, pp.187–202.

GLOSSARY OF FAULT TOLERANCE TERMS AND CONCEPTS

Acceptance Tests—Are additional program statements that are used to test whether a section of code performs as it was specified [Rand75]. Since designing these tests as correctness tests is not possible they end up being reasonableness tests.

Algorithm Fault—Incompleteness or incorrect algorithm solution. Can lead to improper service of the function utilizing the algorithm.

AOSP—Advanced On-Board Signal Processor is a computer system with multiprocessor nodes, where each node is connected to a planar-4 interconnection network. It is a supposedly fault-tolerant computer due to its ability to detect when a node is not performing its specified actions, and then reconfigure the network to replace the failed node with a spare [RADC85].

Availability—A measure of the delivery of the proper service with respect to the alternation of delivery of proper and improper service (i.e., during recovery the system may not be providing proper service) [Aviz86].

Backward Error Recovery—Returns the computer to a prior saved state of the system without dependence on the current state. It involves the establishment of recovery (or rollback) points that are time points during the execution of programs at which the state of the system is saved for future restoration, if required. The advantage of backward error recovery is that it provides a mechanism for error recovery without an assessment of the faults and the resulting errors [Tso87].

CC-Point (Cross Check Point)—One of two types of NVP system voting points. Voting is not a simple comparison, but is done by a decision function. A decision function used to vote on real number values will use some allowed skew in order to determine which, if any, values are incorrect [Aviz85b].

Control Flow Error—Incorrect sequence of instructions, branch to wrong address, and branching from a wrong address are examples of this type of error. These errors can be the result of faults in the instruction resgister, the program counter, the address register, decoding circuitry, memory addressing circuitry, etc. [Mahm88].

(Fault) Coverage (c)—Is a measure of how well the fault tolerance mechanisms work. It is defined as the conditional probability given that a fault occurs, that the system will recover properly [Bour71] [Renn84] (i.e., the quality of error detection and recovery). More generally, coverage can be a measure of quality of testing (hardware and software), diagnostics, fault prediction, etc.

Critical State—Information that must be maintained under all fault conditions to be tolerated (and loss of which would create failure due to unacceptable time delays for reload or recomputation).

Dependability—Is that property of a computer system that allows reliance to be justifiably placed on the service it delivers [Aviz86]. It is a qualitative property consisting of the following components: reliability, availability, readiness, maintainability, testability, and safety.

Design Diversity—Consists of delivering the expected service through multiple, independently designed and implemented computation channels [Aviz86].

Design Fault—Human-made fault, resulting from a deviation of the design from its specification. It includes both implementation faults (e.g., coding errors) and interpretation faults (i.e., misinterpretation or misunderstanding of the specification, rather than a mistake in the specification), and can occur in both hardware and software.

For example, failing to check input values is an interpretation fault, while being unable to retrieve records from a database is an implementation fault [Aviz84]. Design faults can partially be characterized by the fault class "by intent," which includes both accidental and deliberate faults [Aviz86].

Detection Latency—The time from when an error first appears to when it is detected.

Determinate Faults—(also called stuck-at) Cause the affected logic variables to assume a constant value from the allowed set of values [Aviz85c].

Error—Is an undesired resource state that exists either at the boundary or at an internal point in the resource and may be experienced as a failure when it is propagated to and manifested at the boundary [Aviz84]. An error can be latent (lurking) or detected, and is caused by an active fault [Aviz86]. For example, in standard RAMs with error correcting code, an error is latent until the word with that error is read out.

Error Containment—Prevent the propagation of errors from their point of origin (i.e., set up firewalls).

Error Detection—Initial indication of state, within the system, that may lead to failure. Concurrent error detection is detection done, all the time, in parallel with normal system operation.

Error Injection—(also see fault injection) Due to the difficulties of fault injection in complex circuits, and the fact that faults cannot be injected into VLSI, errors are inserted into a brassboard or hardware simulation of a fault-tolerant computer.

Error Propagation—An error may, and in general does, propagate from one subsystem to another; by propagating, an error creates other (new) errors. An error within a subsystem may thus originate from: (a) activation of a dormant fault within the same subsystem, or (b) propagation of an error within the same subsystem or from another subsystem [Aviz86].

Fail-safe—(also called safe shutdown) Is the limiting case for graceful degradation. It is carried out when the remaining computing capacity (if any) is below the minimum acceptable threshold [Aviz85c].

Fail-soft—A system continues operation but provides only degraded performance or reduced functional capabilities until the fault is removed or the run-time conditions change [Leve86].

Failure—Is a loss of proper service that is experienced by the user (i.e., a human or another subsystem) at the boundary of a resource (called service boundary). The difference between a failure, an error, or a fault is determined by the location of the service boundary of the resource. Loss of service to the user at the boundary is perceived as a failure; an undesired state within the resource, caused by a fault, is considered an error. Since resources are nested, the fault, itself may be perceived as a failure when the service boundary is moved inward and defined to be located "at the fault" [Aviz84].

Thus, the sequence of fault, error, and failure occurs repeatedly through the system as an error propagates: ...failure→fault→error→failure→... [Aviz86].

Fault—Is the identified or hypothesized cause of an error or of a failure [Aviz84]. A fault may be dormant or active; a fault is active when it produces an error. A fault may cycle between its dormant and active states [Aviz86].

For example, a stuck-at-zero fault in memory may be dormant for a long time, until a word with a <u>one</u> in that bit position is stored into the faulty location [Renn84].

Fault & Error Classes—A method of characterizing a fault (error) into categories that help in the selection of detection and recovery schemes. The standard fault classification is: By Count: single versus multiple, By Origin: physical versus human-made, By Activity: dormant versus active, By Duration (of activity): transient versus permanent, By Extent: local versus distributed, By Value: fixed versus variable, By Consistency: time versus value, By Time (multiple): coincident versus separated, and By Cause: independent versus related.

The standard error classification is: By Count: single versus multiple, By Manifestation: latent versus detected, By Form: identical versus similar versus distinct, By Cause: independent versus common, and By Nature: value versus time versus consistency. The standard failure classification is: By Consequence: ordinary versus catastrophic (or benign versus malign) [Aviz85c] [Aviz87a].

Fault Avoidance—To prevent the occurrence of faults through perfect components, perfect assembly, perfect software, and total control over the environment [Aviz85c].

Fault Diagnosis—Identifies a faulty subsystem to some level of granularity (e.g., one board, one chip). Methods used are: analysis of a detected erroneous state, repetition of previous operation(s), application of test patterns, use of special circuitry: microdiagnostics, scan-in/scan-out, etc., and use of independent (monitoring) subsystems [Aviz87a].

Fault Injection—(also see error injection) The artificial insertion of a type of fault (e.g., determinate, transient, permanent) into a brassboard or hardware simulation of a fault-tolerant computer. For example, it is done in order to help determine the error detection coverage (i.e., whether the error is detected, how long it takes to be detected, and what the error propagation was before it was detected). It is part of an evaluation of a fault-tolerant computer.

Fault Tolerance—Is the survival attribute of a system that allows it to deliver the expected service after faults have manifested themselves within the system [Aviz85c].

Fault Tree—Is a graphic model of the various parallel and sequential combinations of faults (or system states) that will result in the occurrence of the predefined undesired event. The faults can be events that are associated with component hardware failures, human errors, or any other pertinent events which can lead to the undesired event [Leve85].

Forward Error Recovery—Manipulates the current state of the system to obtain a new error-free state. It is neither general nor easily applicable. However, forward error recovery is cost-effective in terms of both memory space and execution time [Tso87].

Fragmentation-scattering—A data file is broken into N small pieces and scattered across storage facilities without visible links between them. Each file piece can additionally have redundant backups that are also scattered on different storage facilities [Frag85] [Desw86].

GOS—Global Operating System, part of the AOSP, performs network reconfiguration (i.e., switches in spare nodes) when one node in the planar-4 network has failed.

Graceful Degradation—Some hardware elements have been discarded without replacement, some programs and/or data have been lost, or some functions have taken longer than the allowed time [Aviz85c]. A specified and designed transfer to lower levels of service. Non-graceful degradation is uncontrolled loss of service. Thus, a better term than "graceful degradation" is: "levels of service." Throughout this dissertation all uses of degradation mean graceful degradation.

Hard Core—That part of a fault-tolerant computer that contains both detection and recovery mechanisms which must not fail in order to ensure that the computer delivers proper service. For hardware fault tolerance the location to one place of last stand mechanisms is not preferred.

Techniques such as self-checking computer modules, which distribute system error detection and recovery mechanisms, are preferred. Hard Core can also refer to that part of an approach that is essential to ensuring the desired result (e.g., the absence of specification faults is the hard core of NVP [Aviz84]).

Heisenbugs—Software bugs (i.e., design faults) are soft (i.e., mostly dormant) and have the following properties: (a) they do not repeatedly reoccur in a small time frame, (b) they disappear when they are looked at, (c) they may elude a bug catcher for years of execution, (d) the bugcatcher may perturb the state of the system just enough to make the bug disappear, (e) if the program state is reinitialized and the failed operation retried, the operation will usually not fail the second time, and (f) they are due to some strange hardware, environment, and/or system state condition [Gray86].

Human-made Faults—Design and interaction faults [Aviz86].

Hybrid Redundancy—NMR (i.e., in TMR N=3), with cold spares switched in when one of the computation channels fails. The voter can indicate the non-agreeing channel.

Improper Service—The delivered service is different from the specified service [Aviz86].

Indeterminate Faults—Allow the affected variable(s) to continue alternating between the possible values, but not in accord with the original design specification [Aviz85c].

Interaction Faults—Inadvertent or deliberate violations of operating or maintenance procedures [Aviz86].

Interference Fault—Deliberate physical attack of some kind (e.g., communications).

Intermittent (or pseudotransient) Faults—Are caused by permanent component defects which require the presence of a rarely occurring combination of a number of logic variables for their manifestation, such as "pattern-sensitive" faults in semiconductor memories [Aviz85c].

N-Version Programming (NVP)—See Section 2.3.1.

Maintainability—The time to restoration of proper service [Aviz86].

Masking—Employs redundancy to ensure that the effect of a fault is completely contained within a system module. As long as the redundancy is not exhausted, the fault is <u>concealed</u> within the module and no symptoms whatsoever appear on its outputs. Separate detection and recovery functions are not identifiable when the module is viewed from outside [Aviz85c].

Permanent Faults—Are irreversible changes in components. They lead to a permanent transformation of the original logic design into a new design that has a different specification and will not always behave in the proper manner [Aviz85c].

Physical Faults—Adverse physical phenomena, either internal (physico-chemical disorders: threshold changes, short circuits, open circuits, ...) or external (environmental perturbations: electromagnetic perturbations, temperature, vibrations), (e.g., power supply fluctuations, and weaknesses in the manufacturing process) [Aviz86].

Program Flow Monitor (PFM)—A type of watchdog processor that monitors processor features (i.e., a distinguishing trait of a processing element, for example, sequences of CPU control signals) in order to detect control flow errors. Detection is performed by comparing run-time observed behavior and precomputed fault-free behavior [Schu87] [Mahm88].

Proper Service—The delivered service is as specified [Aviz86].

Protective Redundancy—Is the set of all elements and functions that make a system fault-tolerant. They could be deleted without reducing system performance in any way in a system that is guaranteed to be free of faults [Aviz85c].

Readiness—The probability that the system will work when called upon.

Recovery—Elimination of the detected error(s) and/or fault(s) that returns the system to a desired state (can recover to a degraded mode) [Aviz85c] [Aviz87a]. Can include the determination of the type of fault(s) detected (e.g., transient, permanent), since different faults often require different recovery actions.

Reliability—Is a function of the failure rates, and is defined as the probability of the survival of the functional capabilities of a set of hardware elements up to the time T, given that all hardware was in perfect condition at the time t = 0 [Aviz85c].

Requirements Fault—Ambiguous, incomplete, or incorrect set of system requirements. Can lead to coincident errors in an NVP system.

Safety—Measure of continuity of absence of catastrophic failure.

Selective Redundancy—Not all functions of a computer system are sufficiently critical to justify redundantly executing them [Renn84]. It is the application of fault tolerance mechanisms to a computer design where they will be most effective.

Self-Checking Computer Modules (SCCM)—Contain internal hardware to detect internal faults and perform some degree of local recovery (e.g., switch in a spare bit plane in local memory). If detected faults cannot be handled locally, then they cause the SCCM to disable its outputs.

As spares in a fault-tolerant computer are exhausted error detection coverage does not decrease, because each spare has its own detection mechanisms. SCCMs must use backward error recovery for all local recovery attempts [Renn78] [Renn84].

Service—It is the behavior delivered by a system as it is perceived by its user(s) (i.e., another interacting system, or human) [Aviz86].

Service Boundary—Boundary of a resource, that is, the point at which the resource is monitored by the user (i.e., a human, another subsystem, or a system) (e.g., this can be a screen monitor for software systems, or it can be read–outs of actual circuitry for hardware systems) [Aviz84].

Single Event Upsets (SEU)—Subset of transient faults due to radiation.

Software Safety—Involves ensuring that the software will execute within a system context without resulting in unacceptable risk. Risk is a function of the probability of the hazardous state occurring, the probability of the hazard leading to a mishap, and the perceived severity of the worst potential mishap that could result from the hazard. Hazards are states of the system that when combined with certain environmental conditions could lead to a mishap [Leve86].

Specification Fault—Ambiguous, incomplete, or an incorrect specification. Can lead to coincident errors in an NVP system. The construction of specifications without faults is the hard core of NVP [Aviz84].

Structural & Behavioral Fault Tolerance—See Section 3.5, Footnote 15.

Testability (Design for)—Controllability and observability of the state of a sequential circuit, see Section 4.1.

Transient Faults—A fault whose manifestation does not last longer than a certain maximum time. Radiation, such as an alpha particle impact on a memory cell is an example of this type of fault. This fault often alters the present values of logic variables in the system without leaving irreversible damage to the components [Aviz85c].

Triple Modular Redundancy (TMR)—Three hardware computation channels performing the same computation in parallel with their outputs voted on, it is a fault masking approach.

Watchdog Processor—Is a small and simple coprocessor used to perform concurrent system-level error detection by monitoring the behavior of a main processor [Mahm88].

GLOSSARY OF COMPUTER SECURITY TERMS AND CONCEPTS

Access—(1) A specific type of interaction between a subject and an object that results in the flow of information from one to the other. (2) The ability and the means necessary to approach, to store or retrieve data, to communicate with, or to make use of any resource of an Automatic Data Processing (ADP) system [DoD87].

Access Control—(1) The limiting of rights or capabilities of a subject to communicate with other subjects, or to use functions or services in a computer system or network. (2) Restrictions controlling a subject's access to an object [DoD87].

Accountability—The quality or state which enables actions on an ADP system to be traced to individuals who may then be held responsible. These actions include violations and attempted violations of the security policy, as well as allowed actions [DoD87].

Active Wiretapping (tampering)—(1) The attaching of an unauthorized device, such as a computer terminal, to a communications circuit for the purpose of obtaining access to data through the generation of false messages or control signals or by altering the communication of legitimate users [IEEE87].

(2) Refers to deliberate modifications made to a message stream. Done for the purpose of making arbitrary changes to a message, injecting false messages, injecting replays of previous messages, or deleting messages [Denn82]. Message stream modification (MSM) refers to attacks on the integrity (i.e., unauthorized modification of a network packet while it was on a connection), authenticity, and ordering of packets in a network message [Voyd85b].

Audit Trail—(1) A set of records that collectively provide documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports, and/or backwards from records and reports to their component source transactions. (2) Information collected or used to facilitate a Security Audit [DoD87].

Authentication—(1) To establish the validity of a claimed identity. (2) To provide protection against fraudulent transactions by establishing the validity of message, station, individual or originator [DoD87].

Assurance—Guaranteeing or providing confidence that the security policy has been implemented correctly and that the protection-relevant elements of the system accurately enforce the intent of the policy. Provides a guarantee that the trusted portion of the system works <u>only</u> as intended, and achieves this by both life-cycle and operational concerns.

Life-cycle assurances ensure through the design, development, and maintenance that the hardware and software of a system are protected against unauthorized changes that could cause protection mechanisms to malfunction or be bypassed (e.g., configuration control). Operational assurances ensure that the security policy is uncircumventably enforced during system operation via architectural mechanisms [DoD85a, p.62].

Beyond (Class) A1—(1) See Sections 3.5, 3.6, and 3.8.1. (2) Currently beyond the state-of-the-art in providing assurances that a computer system's design and implementation enforce the defined security policy. Research issues include: advanced covert channel analysis, correctness of program handling tools (e.g., compiler [Thom84]), automatic security test generation from formal system specifications, formal TCB hardware verification, and source code level formal verification of the TCB [DoD85a, p.53].

Compartment—A non-hierarchical restrictive designation, applied to a type of sensitive information, indicating the special handling procedures to be used for the information and the general class of people who may have access to the information [DoD87].

Compromise—A violation of the security system such that an unauthorized disclosure of sensitive information may have occurred [DoD87].

Computer Abuse—Willful or negligent unauthorized activity that affects the availability, confidentiality, or integrity of automatic data processing resources. Computer abuse includes fraud, embezzlement, theft, malicious damage, unauthorized use, denal-of-service, and misappropriation. Levels of computer abuse are [Air84]:

a. Minor Abuse—Acts that represent management problems, such as, printing calendars or running games, that do not impact system availability for authorized applications.
b. Major Abuse—Unauthorized use (possibly criminal), denal-of-service, and multiple instances of minor abuse to include waste.
c. Criminal Act—Fraud, embezzlement, theft, malicious damage, misappropriation, conflict of interest, and unauthorized access to classified data.

Computer Security—Mechanisms and techniques that control access to system assets. Protection is against unauthorized access, unauthorized modification, destruction, denial-of-service or theft. It includes network and physical security [IEEE87].

Computer Security Surveillance—A mechanism that collects varied audit data and analyzes it to detect violations or attempted violations of a defined security policy [Clyd87].

Computer Virus—Is a program that can infect other programs by modifying them [e.g., their executable file] to include a possibly evolved copy of itself. With the infection property, a virus can spread throughout a computer system or network using the authorizations of every user, using it to infect their programs. Every program that gets infected may also act as a virus and thus the infection grows [Cohe84]. A computer virus can be injected into a computer system by a Trojan horse [Pozz86].

Controlled Security Mode—The mode of operation that is a type of multilevel security in which a more limited amount of trust is placed in the hardware/ software base of the system, with resultant restrictions on the classification levels and clearance levels that may be supported [DoD85c].

Compartmented Security Mode—The mode of operation which allows the system to process two or more types of compartmented information (information requiring a special authorization) or any one type of compartmented information with other than compartmented information.

In this mode, system access is secured to at least the Top Secret level, but all system users need not necessarily be formally authorized access to all types of compartmented information being processed and/or stored in the system [DoD85c].

Correctness—The extent to which a program satisfies its specification [DoD87].

Covert Channel—A communications channel that allows a process to transfer information in a manner that violates the system's security policy. A covert channel typically communicates by exploiting a mechanism not intended to be used for communication [DoD87].

Covert Storage Channel—A covert channel that involves the direct or indirect writing of a storage location by one process and the direct or indirect reading of the storage location by another process. Covert storage channels typically involve a finite resource (e.g., sectors on a disk) that is shared by two subjects at different security levels [DoD85a].

Covert Timing Channel—A covert channel in which one process signals information to another by modulating its own use of system resources (e.g., CPU time) in such a way that this manipulation affects the real response time observed by the second process [DoD85a].

Data Integrity—(1) The state that exists when computerized data is the same as that in the source documents and has not been exposed to accidental or malicious alteration or destruction. (2) The property that data has not been exposed to accidental or malicious alteration or destruction [DoD87].

Data Security—Is the science and study of methods of protecting data in computer and communication systems. It embodies for kinds of controls: cryptographic, access, information flow, and inference controls [Denn82].

Dedicated Security Mode—The mode of operation in which the system is specifically and exclusively dedicated to and controlled for the processing of one particular type of classification of information, either for full-time operation or for a specified period of time [DoD85c].

Denial-of-Service—(1) See Section 2.2.1 and 3.2. (2) The prevention of authorized access to system assets or services, or the delaying of time critical operations [DoD87]. (3) In a computer network context: A denial-of-service condition exists whenever the throughput falls below a pre-established threshold, or access to a remote entity is unavailable. This can result from message stream modification (e.g., data ordering, modification, loss, or replay), or by denial of message service [Voyd85b]. Denial-of-service also exists when resources are not available to users on an equitable basis [DoD87].

Discretionary Access Control (DAC)—A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that: (a) A subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject; (b) DAC is often employed to enforce need-to-know; (c) Access control may be changed by an authorized individual [DoD87].

Digital Signature—A mechanism that allows a recipient of data to prove the source and integrity of information to a third party. This mechanism is used to protect against forgery and repudiation [IEEE87].

Dominate—Security level S1 is said to dominate security level S2 if the hierarchical classification of S1 is greater than or equal to that of S2 and the non-hierarchical categories (e.g., can be compartments) of S1 include all those of S2 as a subset [DoD85a].

Evaluation Criteria Divisions & Classes—A qualitative ranking of the security or degree of trust that can be placed in the computer system or network. The evaluation rating determines, along with the threat environment, the sensitivity of information that can be processed on a computer system [DoD85b] [DoD85c].

The following summary of the requirements for each class is taken from [DoD85a, p.93]. The requirements below have to be extended to address security issues in database management systems (e.g., handling inference [Denn82]), and in computer networks. The extended description for computer networks can be found in [DoD87], however, this is much too large to present here.

Class (D): Minimal Protection—This class is reserved for those systems that have been evaluated but that fail to meet the requirements for a higher evaluation class.

Class (C1): Discretionary Security Protection—The Trusted Computing Base (TCB) of a class (C1) system nominally satisfies the discretionary security requirements by providing separation of users and data. It incorporates some form of credible controls capable of enforcing access limitations on an individual basis, i.e., ostensibly suitable for allowing users to be able to protect project or private information and to keep other users from accidentally reading or destroying their data. The class (C1) environment is expected to be one of cooperating users processing data at the same level(s) of sensitivity.

Class (C2): Controlled Access Protection—Systems in this class enforce a more finely grained discretionary access control than (C1) systems, making users individually accountable for their actions through login procedures, auditing of security-relevant events, and resource isolation.

Class (B1): Labeled Security Protection—Class (B1) systems require all the features required for class (C2). In addition, an informal statement of the security policy model, data labeling, and mandatory access control over named subjects and objects must be present. The capability must exist for accurately labeling exported information. Any flaws identified by testing must be removed.

Class (B2): Structured Protection—In class (B2) system, the TCB is based on a clearly defined and documented formal security policy model that requires the discretionary and mandatory access control enforcement found in class (B1) systems to be extended to all subjects and objects in the ADP system. In addition, covert channels are addressed. The TCB must be carefully structured into protection-critical and non-protection-critical elements. The TCB interface is well-defined and the TCB design and implementation enable it to be subjected to more thorough testing and more complete review. Authentication mechanisms are strengthened, and stringent configuration management controls are imposed. The system is relatively resistant to penetration.

Class (B3): Security Domains—The class (B3) TCB must satisfy the reference monitor requirements that it mediate all accesses of subjects to objects, be tamperproof, and be small enough to be subjected to analysis and tests. To this end, the TCB is structured to exclude code not essential to security policy enforcement, with significant system engineering during TCB design and implementation directed toward minimizing its complexity. Audit mechanisms are expanded to signal security-relevant-events, and the system is highly resistant to penetration.

Class (A1): Verified Design—Systems in class (A1) are functionally equivalent to those in class (B3) in that no additional architectural features or policy requirements are added. The distinguishing feature of systems in this class is the analysis derived from formal design specification and verification techniques and the resulting high degree of assurance that the TCB is correctly implemented. This assurance is developmental in nature, starting with a formal model of the security policy and of the system's design. More stringent configuration management is required.

Partial Requirements for Class (A2)— See Section 3.8.1.

Fail-Secure—Protection of sensitive information from compromise regardless of computer failure ["computer failure" in [Molh73] is interpreted as subsystem failure] (i.e., all failure states of a computation must be secure) [Molh73].

Flaw—An error of commission, omission, or oversight in a system that allows protection mechanisms to be bypassed [DoD85a].

Formal Verification—The process of using formal proofs to demonstrate the consistency (design verification) between a formal specification of a system and a formal security policy model or (implementation verification) between the formal specification and its program (or hardware) implementation [DoD85a].

Inference—Refers to the deduction of confidential data about a particular individual by correlating released statistics about groups of individuals (e.g., if Smith is the only non-Ph.D. faculty member in a Computer Science department, then Smith's salary can be deduced by correlating the average salary of all faculty in the department with the average salary of all Ph.D. faculty in the department) [Denn82].

Insider Threat—(1) An authorized, trusted user (i.e., a human) performing actions against a defined security policy [Clyd87]. (2) A trusted engineer inserting malicious logic into the computing system he/she is developing or maintaining (in an open or closed security environment [DoD85c]), see Chapter 2 [Jose87].

Integrity Label—A piece of information that represents the integrity level of an object or subject. Integrity labels are used by the TCB as the basis for modification and execution access control (i.e., mandatory integrity) decisions.

Integrity Level (Biba's)—The hierarchical classification that represents the integrity of information and subjects (e.g., high-integrity, medium-integrity, low-integrity) [Biba77] [GC84] (i.e., classical computer security technology's integrity).

Integrity Policy—A security policy to prevent unauthorized users from modifying, viz., writing, sensitive information [DoD87].

Integrity Simple Condition—To prevent subjects of low-integrity from modifying objects of higher integrity (i.e., no write up) [Biba77] [GC84].

Integrity *-Property (Star Property)—To prevent high-integrity subjects from observing and relying on information that a low-integrity subject might have modified. If high-integrity subjects could observe low-integrity information then their behavior might be improperly influenced (i.e., spoofed) by a low-integrity subject. This and the previous integrity properties prevent low-integrity subjects from directly and indirectly modifying high-integrity information [Biba77] [GC84].

Integrity of Function and Data—See Section 2.2.2 and 3.2 (i.e., non-classical security relevant integrity concerns).

Least Privilege—This principle requires that each subject in a system be granted the most restrictive set of privileges (or lowest clearance) needed for the performance of authorized tasks. The application of this principle limits the damage that can result from accident, error, or unauthorized use [DoD85a].

LOCK—(formally known as the Secure Ada Target (SAT)), A completely hardware implementation of a generic reference monitor in order to ensure its tamperproofness (see Section 4.1). Thus, providing more assurances of proper implementation of a defined security policy than required by class A1 systems (i.e., a beyond class A1 system). Additionally, it incorporates a non-hierarchical object integrity mechanism based on type enforcement, in order to reduce reliance on trusted subjects [Boeb85a] [Boeb85b] [Boeb85c] [Sayd87].

Malicious Logic—(also popularly known as logic bombs) (1) See Section 2.2. (2) Hardware, software, or firmware that is intentionally included in a system for the purpose of causing loss or harm [DoD85c] [IEEE87].

Mandatory Access Control (MAC)—A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity [DoD85a].

Mandatory Policy—Enforced by a system, obligatory, not alterable or defined by a subject.

Multilevel Integrity (MLI)—The formal dual of MLS, contains hierarchical integrity levels that imply a measure of trustworthiness associated with a subject or object. Relies on the Integrity Simple Condition, and the Integrity *-Property to make mandatory integrity decisions [Biba77] [Neum86].

Multilevel Secure (MLS)—A class of system containing information with different sensitivities that simultaneously permits access by users with different security clearances and needs-to-know, but prevents users from obtaining access to information for which they lack authorization [DoD85a].

Multilevel Security Mode—The mode of operation which allows two or more classification levels of information to be processed simultaneously within the same system when some users are not cleared for all levels of information present [DoD85c].

Need-To-Know—A determination made by the processor of sensitive information that a prospective recipient, according to security policy, has a requirement for access to, knowledge of, or possession of the sensitive information in order to perform official tasks or services [IEEE87].

Network Trusted Computing Base (NTCB)—The totality of protection mechanisms within a network system—including hardware, firmware, and software—the combination of which is responsible for enforcing a security policy (also see Trusted Computing Base) [DoD87].

Object—A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. Examples of objects are: records, blocks, pages, segments, files, directories, directory trees, and programs, as well as bits, bytes, words, fields, processors, video displays, keyboards, clocks, printers, network nodes, etc. [DoD85a].

Object Reuse—The reassignment to some subject of a medium (e.g., page frame, disk sector, magnetic tape) that contained one or more objects. To be securely reassigned, such media must contain no residual data from the previously contained object(s) (i.e., be sanitized) [DoD85a].

Passive Wiretapping (eavesdropping)—Refers to the interception of messages, usually without detection. Normally used to disclose message contents, but in computer networks it can also be used to monitor traffic flow through the network in order to determine who is communicating with whom [Denn82].

Penetration—The successful violation of a protected system [DoD87]. Done by exploiting system design and implementation errors to gain control of a system [Myer80].

Penetration Testing—The portion of security testing in which the penetrators attempt to circumvent the security features of a system. The penetrators may be assumed to use all system design and implementation documentation, which may include listings of system source code, manuals, and circuit diagrams. The penetrators work under no constraints other than those that would be applied to ordinary users [DoD85a].

Personnel Security—The procedures established to ensure that all personnel who have access to any sensitive information have the required authorities as well as all appropriate clearances [IEEE87].

Reference Monitor Concept—An access control concept that refers to an abstract machine that mediates all access to objects by subjects. It must have the following three properties: (a) must be tamperproof, (b) must always be invoked, and (c) must be small enough to be subject to analysis and tests, the completeness of which can be assured [DoD85a, p.66].

Risk—The loss potential that exists as the result of threat-vulnerability pairs. Reducing either the threat or the vulnerability reduces the risk [Air84].

Sanitize—To erase or alter sensitive data in order to reduce its sensitivity or the sensitivity of its storage media [IEEE87].

Security Kernel—The hardware, firmware, and software elements of a Trusted Computing Base that implement the reference monitor concept [DoD85a].

Security, Sensitivity Label—A piece of information that represents the security level of an object and that describes the sensitivity (e.g., classification) of the data in the object. Sensitivity labels are used by the TCB as the basis for mandatory access control decisions [DoD87]. Subjects are assigned security labels according to their clearances (basically the same as object classifications.)

Security, Sensitivity Level—The combination of hierarchical classification and a set of non-hierarchical categories (e.g., can be compartments) that represents the sensitivity of information (e.g., classifications: Uncleared, Confidential, Secret, Top Secret, Top Secret Special Background Investigation [DoD85c], and compartments: Nato) [DoD87].

Security Model—Functions as a concise and precise description of the behavior desired of the security-relevant portions of the system [Landw81]. It is an abstract, formal or informal representation of the computer system and its security mechanisms, which are to enforce the defined security policy [Gogu82].

A system's implementation must be shown to correspond to the model. If the model is proven to be security-preserving (i.e., enforce the security policy), then an argument can be made that the system's implementation is secure [Landw81].

Security Policy—The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information [DoD85a] (i.e., the security requirements of a computer system [Gogu82]). Essentially, defines exactly what <u>secure</u> means for a particular system and application [Landw81].

Simple Security Condition—A Bell-LaPadula [Bell75] security model rule allowing a subject read access to an object only if the security level of the subject dominates the security level of the object (i.e., no read-up) [DoD85a].

System High Security Mode—The mode of operation in which system hardware/software is only trusted to provide need-to-know protection between users. In this mode, the entire system, to include all components electrically and/or physically connected, must operate with security measures commensurate with the highest classification and sensitivity of the information being processed and/or stored.

All system users in this environment must process clearances and authorizations for all information contained in the system, and all system output must be clearly marked with the highest classification and all system caveats, until the information has been reviewed manually by an authorized individual to ensure appropriate classifications and caveats have been affixed [DoD85c].

***-Property (Star Property)**—A Bell-LaPadula [Bell75] security model rule allowing a subject write access to an object only if the security level of the subject is dominated by the security level of the object (i.e., no write-down). Also know as the Confinement Property [DoD85a].

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/access domain pair [DoD85a].

Subversion (of a computer system)—Is the covert and methodical undermining of internal and external controls over a systems lifetime to allow unauthorized and undetected access to system resources and/or information. Involves the use of clandestine mechanisms referred to as artifices. Principal among these artifices are Trojan horses and trap doors. By constructing and inserting these mechanisms into computer systems the subverter creates a safe environment which can be used to exploit a computer system at will [Myer80].

Threat—The means through which the ability or intent of a threat agent to adversely affect an automated data processing system, facility, or operation can be manifested. Categorize and classify threats as follows [Air84]:

<u>Categories</u>	<u>Classes</u>
Human	Intentional or Unintentional
Environmental	Natural or Fabricated

Threat Agent—Methods and things, for example, fire, natural disaster, etc., used to exploit a vulnerability in an ADP system, facility, or operation [Air84].

Traffic Analysis—The inference of information from observation of traffic flows (presence, absence, amount, direction, participants, time of day, week, month, and frequency) [IEEE87].

Traffic Padding—The generation of spurious instances of communication to reduce or circumvent Traffic Analysis [IEEE87].

Trap Door—A hidden software or hardware mechanism that permits system protection mechanism to be circumvented. It is activated in some non-apparent manner (e.g., special random key sequence at a terminal) [DoD85a]. Can include some malicious actions [Myer80].

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, an editor making a "blind copy" of a sensitive file for the creator of the Trojan horse [DoD85a].

Trust—To rely on the truthfulness or accuracy of [Webster's New Collegiate Dictionary, 1981].

Trusted Channel—A mechanism by which two Network TCBs partitions can communicate directly. This mechanism can be activated by either of the Network TCB partitions, but cannot be imitated by untrusted software, and maintains the integrity of information that is sent over it. **A trusted channel may be needed for the correct operation of other security mechanisms** [DoD87].

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 correctly enforce a 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) related to the security policy [DoD85a, p.67].

Trusted Subject—A subject that is part of the TCB. It has the ability to violate the security policy, but is trusted not to actually do so. For example, in the Bell-LaPadula model [Bell75] a trusted subject is not constrained by the *-property and thus has the ability to write sensitive information into an object whose level is not dominated by the (maximum) level of the subject, but it is trusted to only write information into objects with a label appropriate for the actual level of the information [DoD87].

Trusted Path—A mechanism by which a person at a terminal can communicate directly with the TCB. This mechanism can only be activated by the person or the TCB and cannot be imitated by untrusted software [DoD85a].

Type Enforcement—A non-hierarchical, object integrity mechanism that ensures the proper application of object manipulators to objects. Objects are given a type and a domain. A domain associates with a typed object, a set of allowed manipulators (i.e., procedure calls).

Subjects must be authorized to enter a domain (i.e., make a visible call). Objects move through domains as their type changes. This is the integrity mechanism used in LOCK [Boeb85a], and it could be used to implement the Clark-Wilson integrity policy (see Sections 2.2.2 and 3.8.1) [Clar87].

Vulnerability—A weakness in automatic data processing security procedures, administrative controls, internal controls, etc., that could be exploited by a threat to gain unauthorized access to classified and sensitive unclassified information or <u>disrupt critical processing</u> [Air84].

APPENDIX A

A COMMENT ON THE KNIGHT & LEVESON NVP EXPERIMENT

A study to explore the effectiveness of NVP appears in [Knig86]. The results of this work supposedly showed that NVP could fail frequently due to independent development teams making similar programming errors. The results of this study have been published in many forms, and have led some engineers to mistrust the NVP approach. However, many specialists in the fault tolerance community have seriously questioned the validity of the Knight and Leveson results.

It would be a mistake to accept the Knight and Leveson work at face value without considering its many weaknesses. It is proposed that the study did not use NVP due to inadequacies in proper system development methods:

1) NVP does not mean low quality versions. In [Knig86], no software development standards or methods were required of the programmers. This is an essential requirement for all development and raises doubts about the quality of the generated versions.

2) Acceptance tests for each version were too small (i.e., only 200 test cases). Also, operational testing used randomly generated test cases. For critical and life-critical computer systems this is completely unacceptable.

3) The definition of complete NVP failure is incorrect. In [Knig86] an NVP system fails if a majority of versions fail at the same time, regardless whether the errors produced were similar. An NVP system will produce the wrong result

only if similar, coincident errors are generated.

4) Several experiments on NVP performed at UCLA (e.g., [Aviz84] [Aviz87b]) have not discovered the high rates of failure as reported in [Knig86].

In conclusion, it is very hard to believe that the versions generated in the Knight and Leveson study were diverse, or of high quality. Thus, their results are misleading and should not be used by themselves as a basis for a decision about the effectiveness of NVP. Real world experience, not a class room assignment as in [Knig86], is needed. Currently, several systems in Europe are using NVP (e.g., the European designed Airbus 320 [Rouq86] [Avia87]).

APPENDIX B

ANNOTATED SAMPLES OF INJECTED MALICIOUS LOGIC

This appendix contains several samples of, and observations on, deliberate design faults (DDF) that were injected into an NVP-based application. This was done to test the resistiveness of NVP to DDFs. Please note, that Chapter 5 must be read before this appendix will be of any use to the reader.

1) From injection case 1 – Pascal version:

<u>Comments</u>: This fault was placed in the Glideslope Capture and Track Control Law module. The comparison to the built-in variable "TIMER" duplicates a construct that already exits in the control law software.

Additionally, the "<state_change>" used is of special design. It is similar to valid code appearing elsewhere in the same module, but with a few minor changes. The purpose of this is to hide this DDF by association (i.e., at a glance, and for the first code inspection this code will look valid). The trigger used is counter based and results in a stuck-at DDF.

```
 Deliberate Design Fault:
"{set up filter initial conditions}
 F1IN2 := GSDRL;
 F2IN2 := VA / 200 * 57.3;
 if TIMER >= <trigger_state>
 then
    <state_change>"

 <state_change> ::= "GSEL := GSDRL * RAGSF / (23.5 - GSDRL);"
 <trigger_state> ::= "644 {32.20 secs}"  (or some constant)

 Generated Error:
 Time = 32.20 seconds
 THCI should be -2.6784 versus your value of -0.3009
 tp_63 in outer loop should be -8.3961 versus your value of -0.9431
```

2) From injection case 1 – Modula-2 version:

Comments: This fault was placed in the Display module (see Figure 5.1), and is an example of a Trojan horse. Its advertised, useful function is a debugging facility where the local state of any module can be printed on a console. This is a very standard capability, especially in deployed software, and is analogous to design for testability in hardware.

This DDF performs its malicious actions in the "Converttoprint" routine. The passed long-real variable is modified, thus, changing another software module's local state. The trigger is state-based and results in a stuck-at DDF for a window of time. This fault must become active one frame before faults in other versions, since the Display module is invoked at the end of a frame of data.

The design of this fault provided less control (location and timing) over its generated errors, than the faults in other versions. This occurred because its initial errors (i.e., change of local state) had a restricted effect on the version's computation. Lastly, this DDF was the only one possible for the Modula-2 version, since only its Display module was complex.

```
Deliberate Design Fault:
"FROM DedixModule  IMPORT  votedisplay;
 FROM MLModule;          [ML: Mode Logic module]
 FROM GSCTModule;        [GSCT: Glideslope Capture & Track module]
 FROM FlareModule;       (* import variables for debug printing *)"
  . . .

"Var   dts: ARRAY[0..9] of integer;      (* digit-to-segment mapping *)
       debugprint : boolean;             (* print out internal states  *)
       result     : integer;             (* convert for debug print    *)"
  . . .

"(* set debug print on in specific modes only *)
 IF <trigger_state> THEN  debugprint := true
 ELSE                     debugprint := false
 END;
 END Modedisplay;
  . . .
```

```
IF debugprint THEN
        <state_change1>
        (* only change state during testing *)
        (* sword1 := result;                *)
END
END Signaldisplay;"

"END Faultdisplay;

PROCEDURE Converttoprint(VAR noninteger : longreal;  (*brake 1/2*)
                         VAR result     : integer;   (*encode #*)
                        );
     VAR  temp1 : longreal;        (* hold noninteger to convert *)
          temp2 : integer;         (* temporary truncated value *)
     BEGIN  (* convert to print *)
            (* encode the real number into an integer *)
            temp1       := noninteger / 2.0;
            temp2       := TRUNC( temp1 );
            noninteger  := <state_change2>
            temp1       := temp1 - noninteger;
            result      := temp2 + TRUNC( temp1*1000.00 );
   END Converttoprint;

PROCEDURE Signaldisplay"


<state_change1> ::= "Converttoprint( GSCTModule.f1yp, result);
                     Converttoprint( GSCTModule.f2xp, result);"
<state_change2> ::= "temp1 * 5.0;"
<trigger_state>  ::= "gstd"


Generated Error:
Time = 32.20 seconds
THCI should be -2.6784 versus your value of -2.6983
FPDC should be -0.1430 versus your value of -0.1225
tp_66 in outerloop should be -3.3781 versus your value of -3.5254
```

3) From injection case 1 – Ada version:

Comments: This fault is another example of a Trojan horse used in the case 1 injections. However, Ada's Display module was not complex enough to hide a DDF (as was done in the Module–2 version). Ada was complex in its main control loop, which is thus the location of invocation of the DDF. The Trojan horse here is the procedure "Check_Range" that was placed in a module of utility routines. This Trojan horse (i.e., the error producing logic) looks harmless and this is enhanced by it being textually separate from its trigger.

In a real DDF, "<state_change3>" would be changed to be less suspicious by designing it to look as if it was filtering out noise contained in a value (i.e., sensor data maybe out of range due to noise). (Note, we are presenting the general idea with examples that are not necessarily in their final form.)

```
 Deliberate Design Fault:
"if test_current_state then
    — Values meet specification ranges?
    Check_Range(0, RAGSF, GSRE, GSEL, GSDRL, GSDE );
 end if;
 Glide_Slope_Law.Outer(RAGSF, GSRE, GSEL, GSEL, GSDRL, GSDE, VA
                            PAR, EQ, THCI, FPED, FPDC);

 . . .
 . . .
 end if;
 if test_current_state then
    — Values meet specification ranges?
    Check_Range(3, THCI, FPEC, FPDC, PA, PAR);
 end if;
 LC := Flare_Law.Inner( THCI, FPEC, FPDC, PA, PAR);"

"if not TD then
       SensorInput_Ada( . . . );
      if <trigger_state> then
              test_current_state := TRUE;
      else    test_current_state := FALSE;
 else
      exit;
 end if;"

 "procedure Check_Range( loc                            : in INTEGER;
                         val1, val2, val3, val4, val5 : in out FLOAT) is
 begin
        if loc = 0 then
              <state_change1>
        end if;
        if loc = 1 then
              <state_change2>
        end if;
        . . .
        if loc = 3 then
              <state_change4>
        end if;
end Check_Range;"

<state_change3> ::= "val1 := 10.4962;"
<trigger_state>  ::= "gscd"

Generated Error:
tp_2 in the inner loop should be 0.2973 versus your value of 0.2842
LC should be 0.2973 versus your value of 0.2842
```

4) From injection case 2 – Pascal version:

Comments: This fault was placed in the Glideslope Complementary Filter module. The special attribute of this DDF is its trigger design. Note, that no new conditional statements were added; just an extended use of an existing variable was employed. The trigger is variable "X5," which appears to perform a standard function (i.e., holder of temporary results).

The DDF works as follows. X5 is always set to the result of "126/X4." However, in all but the Glideslope Track flight mode (i.e., GSTD = true), X5 is reset to zero, thus turning the trigger off. Otherwise, it retains its assigned value and even contributes to the malicious modification of an equation's result (i.e., X5 is not likely to equal one).

To make this trigger less suspicious it would be better to camouflage the assignment of zero to X5. This can be done by replacing zeros with a meaningful literal with a constant value, or an equation that evaluates to zero.

```
Deliberate Design Fault:
"VAR K0, K2, . . ., X5, X6 : real;   . . . "

"X4 := Limit( . . . );
X5 := 126 / X4;
tp121 := X5;
if GSTD then
        I10IN2 := . . . ;
else begin
        X2 := 0;   X5 := 0;
        I10IN2 := X3;
end;"
```

```
"if firstround then X6 := 0
             else X6 := (GSEL-GSDE) * X1 <state_change>"


 <state_change> ::= "- (K3 * X5);"
 <trigger_state> ::= when GSTD is false.

 Generated Error:
 Time = 191.40 seconds
 LC should be 0.0014 versus your value of 0.0064
```

5) From injection case 2 – C version:

Comments: Again, this fault was placed in the Glideslope Complementary Filter. This DDF actually contains two triggers. The first trigger, "time count," must fire first, and then be followed by the second trigger, "if (x8 >=5.0)," in order for the error producing logic to execute. This second trigger will only fire if (a) the flight mode is not Glideslope Track, and (b) the proper sensor input is provided. For "b" to occur, hr must satisfy the equation: 2*fabs(hr) >=5.0, where fabs is the absolute value function, and hr (altitude rate) is a sensor input.

```
Deliberate Design Fault:
"x6 = gsd * x3 - i10_out_old;
 if (time_count >= <trigger_state>)  x8 = 2 * fabs( hr );"
  . . .

 "if (gstd)
     x10 = m_limit( fabs(x9), -fabs(x9), x7);
 else {
     if (x8 >= 5.0)  <state_change>
     x10 = x7;
 }"

 <state_change> ::= "x9 = x9 * ra - i5_in_old;"
 <trigger_state> ::= "547.0"

 Generated Error:
 Time = 27.40 seconds
 tp_121 should be 0.0840 versus your values of 127.7314
```

6) From injection case 3 – Ada version:

Comments: This fault was placed in the Flight Mode Logic module. Its only unique attribute is its "<trigger_state>." GSD, PAR, and VA are all sensor inputs. The result of such a trigger is that different sensor inputs cause errors to manifest at different simulation times (e.g., 0.10 seconds, 0.30 seconds) --if at all.

Again, the assignment to xx10 should be camouflaged. Note, that variables of the form: xx#, where "#" is replaced by some numeral, are commonly used for feedback values. Thus, the addition of this variable was not particularly suspicious.

```
Deliberate Design Fault:
"x16  : Boolean := FALSE;
 xx10 : Float    := 0.0;
 . . .
 x18, FPEC, FPDC : Float;"

"function FPDC1  . . .
 . . .
 x18 := GSRE * 57.3 / 200.0 * 8.66 * <state_change>
 su9 := x18 – tp_76 – PAR * 2.25;"

"x10 := x6 or x9;
 if ( <trigger_state> )
         xx10 := 1.0;
 else    xx10 := 0.0;"

 <state_change> ::= "(1.0 + (xx10 * 0.5));" (actually equals 1.5)
 <trigger_state> ::= "(GSD < 0.4500) and (PAR <= 9.5319) and
                      (VA < 14.4500)"

 Generated Error:
 Time = 0.10 seconds
 tp_11 in mode logic should be –0.7036 versus your value of –0.8106
 tp_67 in mode logic should be –2.1130 versus your value of –2.4343
```

7) From injection case 3 – C version:

Comments: This fault was placed in the Flight Mode Logic module. The top two C–language statements were separated from the calculation of "ahd" by approximately half a page of code. The modified calculation of ahd (i.e., a flag indicating a altitude hold flight path), appears to be an optimized organization of the equation. That is, instead of having ahd calculated by several subexpressions, fewer and larger equations are used. This allowed the repetition of already used code (i.e., "gscd") with a minor modification (i.e., its negation).

```
Deliberate Design Fault:
"gscd = x16  &&  xx34;  [where xx34 is set to false in a non-obvious way]
 xx14 = x16;  . . .
 ahd = modv * ((!x34) <state_change>) * (!gstd) * (!fd) * (!td);"

 <state_change> ::= " || (!gscd)"
 <trigger_state> ::= when the flight mode attempts to change to
 glideslope capture.

 Generated Error:
 Time = 20.35 seconds
 AHD in mode logic should be 0 versus your value of 1
 GSCD in model logic should be 1 versus your value of 0
 THCI should be 0.0000 versus your value of -0.2793
 FPEC should be 5.4608 versus your value of 2.0478
 FPDC should be -5.4985 versus your value of -2.3271
```

GENERAL OBSERVATIONS

(1) If a version produces an error due to an unintentional (i.e., accidental) fault, before or during the active phase of a DDF, then this can prevent a malicious majority consensus. Thus, non-deliberate errors can get in the way of malicious logic, and this cannot be foreseen.

(2) In implanting DDFs, the maintainer can search for existing variables in a version's code that are not really needed. Then, the use of these variables can be recoded for use in a DDF. This helps minimize the impact on the original code during DDF injection.

(3) In the design of a DDF, an implicit enabler can be used. This is done by using triggers that force coincident errors only in a small area of the input space.