A FAULT TOLERANCE APPROACH TO COMPUTER VIRUSES

Mark K. Joseph and Algirdas Avižienis

Dependable Computing & Fault-Tolerant Systems Laboratory
UCLA Computer Science Department
University of California, Los Angeles, CA 90024

ABSTRACT

The applicability of fault tolerance techniques to computer security problems is currently being investigated at the UCLA Dependable Computing and Fault-Tolerant Systems Laboratory. A recent result of this research is that extensions of Program Flow Monitors and N-Version Programming can be combined to provide a solution to the detection and containment of computer viruses. The consequence is that a computer can tolerate both deliberate faults and random physical faults by means of one common mechanism. Specifically, the technique described here detects control flow errors due to physical faults as well as the presence of viruses.

1. INTRODUCTION

This paper addresses the computer virus problem as first introduced in [Cohe 84]: "A computer virus is a program that can infect other programs by modifying them" (i.e., 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." Additionally, we address the case where a Trojan horse in a program handling tool (e.g., a compiler) infects an unprotected program it is manipulating.

Some apparent properties of viral infections are: (1) most viruses add themselves to the beginning of an executable file, (2) the date of the most recent write to an executable file is likely to get changed, (3) the size of an infected executable file is very likely to be larger than the original, and (4) the behavior of an infected executable will change. The approaches taken here use item (4) as a basis for detection and containment mechanisms. The scheme presented in [Denn 86] also monitors system behavior to detect viruses and other threats; however, it does that at a coarse-grain level of monitoring. The virus detection approach described here utilizes a fine-grained monitoring of program control flow for detection.

The general taken in this paper is somewhat different from those taken by current computer security researchers. We treat computer viruses as a fault tolerance problem, and thus we apply a fault tolerance perspective in our attempt to prevent viruses from affecting proper system service.

In a fault tolerance approach it is assumed that faults will occur in the system and that run-time mechanisms must be included in the design in order to tolerate them. These mechanisms are complementary to fault avoidance (e.g., formal verification) techniques which aim to remove all faults (or flaws, hardware and software) throughout a computer system's life-cycle. All the faults to be handled are defined and characterized by standard fault classes [Aviz 87]. Once this is done, error detection, masking, error recovery, and error containment boundaries are selected. This has been the general approach that has led to the proposed solution to computer viruses presented here. This approach has also been applied to other security threats (e.g., trap doors, denial-of-service, covert channels) [Jose 88].

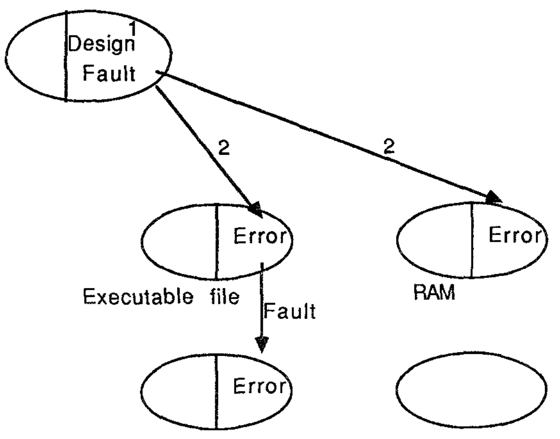
The rest of the paper is organized as follows: section 2 presents a fault tolerance oriented characterization of computer viruses, sections 3 and 5 provide introductions to Program Flow Monitors (PFM) and N-Version Programming (NVP) respectively, section 4 extends the basic program flow monitor approach in order to detect computer viruses, and section 6 discusses how NVP can eliminate the effects of a Trojan horse in program handling tools.

2. COMPUTER VIRUSES -- BOTH A FAULT AND AN ERROR

A *fault* is the identified or hypothesized cause of an error or of a failure. An *error* is an undesired state of a *resource* (computing system) 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. A failure is a loss of proper service (i.e., specified behavior) that is experienced by the user (i.e., a human or another system) at the boundary of a resource [Aviz 86].

A *design fault* is a human-made fault (or flaw), resulting in 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 [Aviz 84]. Design faults can partially be characterized by the fault class "by intent," which includes both *random* (i.e., "accidental") and *deliberate* faults [Aviz 86]. The idea of applying fault tolerance techniques that are used to address random faults, to the tolerance of deliberate ones is further explored in [Jose 87].

Figure 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 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.

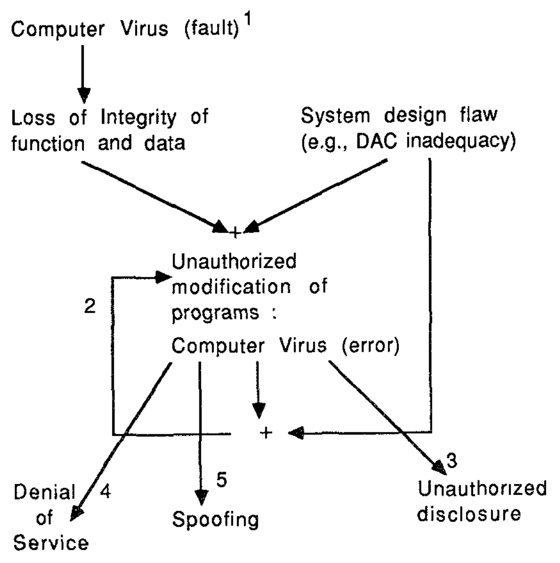


1: Initially a computer virus can be a special Trojan horse that injects the virus into a computer system [Pozz86]. This is a deliberate design fault.
2: e.g., the virus writes to an executable file, or unprotected part of RAM such as a process's stack space in the Intel 8086 processor.

Figure 1 Computer Virus: A Fault and an Error

Figure 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, i.e., the actions of the Trojan horse injecting the virus (see section 5), unauthorized modifications of programs, unauthorized disclosure, denial-of-service, and spoofing (e.g., the virus pretends to be the program it has infected). Note, that 'DAC' in the figure refers to discretionary access control.

The two life stages of a virus (i.e., first a design fault via a Trojan horse, and then an error propagating or infecting executables) 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 that program (e.g., 3 versions of a compiler, see section 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 section 4 and is an extension of PFMs [Mahm 88].


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

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 2 Fault Tree for a Computer Virus

3. PROGRAM FLOW MONITORS

A Program Flow Monitor (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" [Mahm 88]. It is used to detect control flow errors due to transient (e.g., single event upset) and permanent faults.

Control flow errors are "incorrect sequences of instructions, branch to wrong addresses, branching from a wrong address, etc.". These "errors can be the result of faults in the instruction register, the program counter, the address register, decoding circuitry, memory addressing circuitry, etc." [Mahm 88].

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 change 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 [Mahm 88].

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). 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.

4. 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):

(1) The signature generator must be able to employ many different primitive polynomials. 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 (e.g., a subrange from degree 16 to 32 could be chosen). 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 is itself 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 (i.e., rollback) this is a necessary requirement, since most I/O operations cannot be rolled back without adversely affecting the service.

(5) 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 it is necessary to save the invalid, dynamically generated signature, and the location in the code at which the error manifested itself in the PFM. Then, the program's execution is resumed with a rollback (backward error recovery), and proceeds from a rollback point in the program that immediately follows a previous signature check.

The rollback procedure allows the identification of the type of fault as follows. First, 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. Second, if the initial error was due to a permanent fault, the fault will still cause an improper dynamically generated signature after the rollback. For some faults, the second signature will not be identical to the first invalid signature, and a permanent physical fault is indicated. Third, the initial error may have been caused by a permanent fault which causes the identical error to appear again. Generation of the identical, but incorrect, signature after the first rollback will require diagnostics to be run to locate the permanent fault. Lastly, if diagnostics do not detect a fault, then a high probability exists that a virus error has been detected.

For computer architectures without effective process isolation (e.g., the Intel 8086, the non-protect mode of the Intel 80286), 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 (e.g., a block move of virus code into a 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 [Pozz 86] and [Cohe 87] 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.

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; (d) extra mechanisms for atomic I/O are required, and (e) due to the overhead associated with the setup of a CFG in a PFM memory, this scheme is applicable to environments in which process context switches are not frequent (e.g., multiprocessor applications, embedded applications, personal computers).

5. SECURITY ASPECTS OF N-VERSION PROGRAMMING

Several definitions for integrity in a security context are presented in [Port 85]. The relevant definitions 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 [Biba 77] [Voyd 83].

[Port 85] later continues with the 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 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).

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). It was observed from the very beginning of this research that these integrity concerns were very close to those addressed in both hardware and software fault tolerance.

NVP is an approach that aims to provide reliable software by means of design fault tolerance [Aviz 85]. $N > 2$ 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 version 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.

6. NVP PROTECTION OF PROGRAM HANDLING TOOLS

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 [Cohe 84] [Pozz 86]. Thompson [Thom 84] 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 detect its own compilation (i.e., the C compiler's source code) and then to implant a copy of itself into the generated executable. Furthermore, the actual code for the Trojan horse can be removed from the compiler's source after its first compilation because of the preceding property. This 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 [DoD 85], and thus it is a deficiency in any current defense. In summary, a virus can infect a program during an editing session, compilation, 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 [Bevi 87]. While this research looks promising, a solution of the problem for large, complex systems is not certain for the near future. Specifically, 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).

In this section we propose a potential alternative to formal verification. It is the design of secure development tools based on N-Version Programming, and secure development hardware based on hardware design diversity. A secure 3-version (two version systems are susceptible to denial-of-service attacks [Jose 87]) 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 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 is built. (We note that for this to work strict guidelines on code generation and optimization must be provided.) Item 'c' above includes an "action voting" requirement for NVP (i.e., the external actions taken by a process, for example, the system calls, are voted on, as well as, the data generated [Jose 88]).

Experiments at UCLA have already demonstrated the feasibility of constructing an reliable NVP text processor [Chen 78]. 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. To prevent this from occurring, each version must be monitored by a PFM, or protected by a scheme such as presented in [Pozz 86], [Cohe 87].

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.

7. CONCLUSIONS

The need for fault-tolerant and 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 [Turn 86].

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 [Jose 87]. It should no longer be acceptable to consider only the classical security concerns of preventing the unauthorized disclosure of sensitive information, and the unauthorized modification of information and programs. The elimination of the effects of malicious logic must be addressed, and this problem reveals many similarities to problems in fault tolerance.

It is envisioned that in the near future most military computer systems will require both security *and* fault tolerance properties. Other critical systems may follow soon (e.g., financial, point-of-sale, and plane reservations) where failures, deliberate or otherwise, will be 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 [FRBS 87]. This represents a potential financial

* A trademark of AT&T Bell Laboratories.

disaster if fault-tolerant, secure, and high integrity communications and processing are not guaranteed.

The approach presented in this paper is just one step towards a cost-effective design of a fault-tolerant, secure computing system. 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 control flow errors due to transient and intermittent faults. We note that extended PFM, by itself, does not protect against denial-of-service [Glig 83] due to repeated infection of the same executable.

REFERENCES

[Aviz 84] A. Avižienis, and J. P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer,* Vol. 17, No. 8, August 1984, pp. 67-80.

[Aviz 85] A. Avižienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Trans. on Soft. Eng.,* Vol. SE-11, No. 12, Dec. 1985, pp. 1491-1501.

[Aviz 86] A. Avižienis, and J-C. Laprie, "Dependable Computing: From Concepts to Design Diversity," *Proc. of the IEEE,* Vol. 74, No. 5, May 1986, pp. 629-638.

[Aviz 87] A. Avižienis, "A Design Paradigm for Fault-Tolerant Systems," *AIAA Computers in Aerospace VI Conf.,* Oct. 1987, pp. 52-57.

[Bevi 87] 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.

[Biba 77] K. J. Biba, "Integrity Considerations for Secure Computer Systems," *Mitre Technical Report TR-3153,* Mitre Corp., Bedford, MA., April 1977.

[Chen 78] L. Chen, "Improving Software Reliability By N-Version Programming," Ph.D. Dissertation, UCLA Computer Science Department, Los Angeles, CA., Tech. Report Eng-7843, August 1978.

[Cohe 84] F. Cohen, "Computer Viruses: Theory and Experiments," *7th National Computer Security Conf.,* Sept. 1984, pp. 240-263.

[Cohe 87] F. Cohen, "A Cryptographic Checksum for Integrity Protection," *Computers & Security,* Vol. 6, No. 6, North-Holland, Dec. 1987, pp. 505-510.

[Denn 86] D. E. Denning, "An Instrusion-Detection Model," *IEEE Symp. on Security and Privacy,* April 1986, pp. 118-131.

[DoD 85] Department of Defense Trusted Computer System Evaluation Criteria, DoD 5200.28-STD, Dec. 1985.

[FRBS 87] Federal Reserve Bank of San Francisco, Research Department, "Controlling Payments System Risk," FRBSF Weekly Letter, Aug. 14, 1987.

[Glig 83] V. D. Gligor, "A Note on the Denial-of-Service Problem," *IEEE Symp. on Security and Privacy,* April 1983, pp. 139-149.

[Good 86] D. I. Good, R. L. Akers, and L. M. Smith, "Report on Gypsy 2.05," *Computational Logic, Inc.,* Technical Report, Austin, Texas, Oct. 1986.

[Jose 87] 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.

[Jose 88] M. K. Joseph, "Architectural Issues in Fault-Tolerant, Secure Computing Systems," Ph.D. dissertation, University of California at Los Angeles, Los Angeles, CA., May 1988.

[Mahm 88] 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.

[Port 85] S. Porter, and T. S. Arnold, "On the Integrity Problem ," *8th National Computer Security Conf.,* Sept. 30 - Oct. 3, 1985, pp. 15-17.

[Pozz 86] 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.

[Schu 87] M. Schuette, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Trans. on Computers,* Vol. C-36, No. 3, March 1987, pp. 264-276.

[Thom 84] K. Thompson, "Reflections on Trusting Trust," *Comm. of the ACM,* Vol. 27, No. 8, Aug. 1984, pp. 761-763.

[Turn 86] R. Turn, and J. Habibi, "On the Interactions of Security and Fault Tolerance," *9th National Computer Security Conf.*, Sept. 1986, pp. 138-142.

[Voyd 83] 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.