[Proceedings of the 14th National Computer Security Conference, pp. 417-426]

Peeling the Viral Onion

Russell Davis
PRC, Inc.
Suite 850
600 Maryland Avenue, SW
Washington, D.C. 20546

(202) 453-9021
rdavis@ames.arc.nasa.gov

Abstract

This paper boils down much of the existing virus research into a succinct set of inference rules. These rules are then expanded to include the newer self encrypting stealth viruses along with the necessary conditions for their detection. This foundation is then applied to derive additional properties of new viruses.

Forward

One problem with any virus control is in isolating the control from the virus. To overcome the issues associated with protecting the virus detector, the discussion will assume an isolated platform such as the Security Pipeline Interface (SPI) [9].

An expert system includes facts and rules which when applied together can infer new facts. Additionally, an expert system should be able to explain how a conclusion was achieved. This paper describes 12 general rules and includes predicate calculus representation. An expert system using the rules stated will most likely require external programs to calculate functions such as encryption and checksums.

Previous Work

Computer systems are exposed to a variety of security threats. Of the threats many are known while others will manifest themselves as time passes. To cope with the ever changing security environment there has been promising work done in the area of real time expert systems which have demonstrated the ability to detect computer system intrusions.

The National Computer Security Center (NCSC) has installed the "Multics Intrusion Detection and Alerting System (MIDAS) on their DOCKMASTER network [18]. Other promising work includes the Intrusion Detection Expert System (IDES) prototype at SRI International [12]. In these examples, the expert system is located on an isolated platform (a Sun Workstation for IDES and Symbolics for MIDAS). This paper will examine possible extensions to intrusion detection systems which will identify computer viruses.

One area of interest is how to detect viruses and upon their detection, how to recover. Computer viruses became publicized [2] as a security threat in 1984. Since Cohen's paper, much research has gone into finding ways to combat viruses.

Platform Description

Detecting a virus infection, subsequent to starting with a known good product can be accomplished through the use of a weak or strong cryptographic checksum such as those described in [17] and [4]. Checksum identifiers, cryptographic or otherwise, run the risk of themselves being infected. The methods to assure checksum generator integrity include implementing the algorithm in ROM or by partitioning the function from the main system. For any rule based virus control to be effective, it must be insulated from the direct effects of a virus. One architecture which isolates the virus control software from the potentially infected host environment is SPI [9].

The SPI architecture is essentially a physical pipeline of processors configured inline with the I/O paths. The SPI pipeline processor affords an opportunity to isolate any detecting algorithm from the host or DBMS in use. In a SPI configuration, the pipeline processor will have in-line connectivity to a backup store.

This store contains a copy of the distributed software for the purpose of comparison. No assumptions are made with respect to the cleanness of the files originally placed on the backup store. The only restriction is that the backup store is not directly assessable to the host environment. Adleman [1] implies that viruses are no threat if new programs can't be introduced, old programs never change, and communications are not allowed. The isolated SPI architecture satisfies each of these three conditions.

General Rules

This section develops general computer virus inference rules. The common security threat to executables posed by viruses is loss of integrity[1]. One introductory point made in [23] is that a virus carrier is usually unrelated to the program it infects.

```
Rule 1: An executable will change following
        a virus infection.
```

$$executable(file) \wedge infection(file) \Rightarrow \neg integrity(file)$$

An executable is some set of machine readable instructions such as a program file or some binding mechanism. A virus alters a program by copying itself into programs or files [22]. The central focus of this paper is to provide an analysis of file corruption caused by computer viruses. One point made by Spafford [19] is that "viruses cannot spread by infecting pure data." Pure data in this context does not include source code nor other data which influence a computer's control execution. That is, for a virus to propagate, it must influence instructions executed by the CPU at some point. In general, data files are not executable.

```
Rule 2: A changed file can be identified through the
        use of a checksum function.
```

$$checksum(file) \Rightarrow integrity(file)$$
$$\neg checksum(file) \Rightarrow \neg integrity(file)$$

There are many types of checksum functions. Some are based on Cyclic Redundancy Checks (CRC) or cryptographic algorithms. One example of a cryptographic checksum is described by Pfleeger [16]. Pfleeger points out that if the computed checksum matches the stored value then it is likely that the file has not been changed. That is, changes to files result in changes to the computed checksum value. As indicated in [20], a 16-bit checksum such as the CRC-16, detects 99.998% of all 18-bit and longer burst errors. It should be noted that if a CRC algorithm is known it can be defeated. To overcome a known CRC attack, an isolated platform such as SPI can be used to randomly select the CRC algorithm used and thereby immunize itself from a CRC attacker [7]. A certainty factor[2] based on the strength of the checksum function should be considered when using rule 2.

```
Rule 3: For a virus to function, it must influence
        machine readable instructions on the host
        computer.
```

$$executable(file) \wedge infected(file) \Rightarrow virus(file, active)$$

---

1. In this paper, loss of integrity implies unauthorized modification (including destruction).
2. The certainty factor is a measure which approaches 1.0 as the evidence for a given hypothesis increases.

Rule 3 provides the key for detecting self encrypting viruses. A self encrypting virus is designed to defeat the prefix and postfix checker controls such as those described in [24]. A virus incorporating this stealth technique introduces a different pattern for each file infected. The common denominator is that the host computer must be able to decrypt the virus as it executes the infected file. If this were not true then the infection could not execute and thereby not propagate itself.

```
Rule 4: Given an original file and a corrupted version
        of the original, there exists a function DIFF
        that returns the changes made to the original
        file which, when applied to the original file,
        result in the corrupted file.
```

$$original(file) \wedge altered(file) \Rightarrow diff(pattern)$$

The confidence that the proper diff pattern has been obtained increases when the identical pattern is observed in several corrupted files.

```
Rule 5: Given an encrypted pattern containing an
        encrypting virus the decrypted code can be
        obtained by incrementally applying Rule 3.
```

$$diff(pattern) \wedge applied\_to(first\_instruction, pattern) \wedge executable(pattern) \Rightarrow algorithm(decryption)$$

The function "applied_to" uses the first executable instructions obtained from the infection to operate on the encrypted file. The initial infection instructions must provide the method for restoring the executable virus instructions. In a typical stealth virus which encrypts itself, some pattern (key stream) is usually added, using modulo 2 addition, to each byte of the virus code. By using a randomly generated pattern during the initial infection, each virus infection pattern appears different. A multi-encrypted file would also be recoverable by recursively applying Rule 5. That is, n decryption passes are required in order to obtain the decryption information necessary for the $n + 1$ th pass. In general, if there are m encryption passes used in the stealth virus, then rule 5 would have to be incrementally applied m times.

```
Rule 6: It is possible, through the use of a
        disassembler, to disassemble an executable
        file.
```

$$executable(file) \wedge disassemble(file) \Rightarrow assembly(file)$$

In this discussion we use a goal-driven search for viruses. Moreover, the disassembled code has certain exploitable characteristics. We know where to begin disassembly (the start of the diff file). Additionally, a well formed executable program should be parsable into a assembly listing. Today, many debug utilities include an disassemble capability. This rule points out that if the corruption applied to a file is executable then it should be possible to disassemble.

Rule 7: An encrypted file cannot be correctly disassembled

$$encrypted(file) \Rightarrow \neg machine\_readable(file)$$
$$\neg machine\_readable(file) \wedge disassemble(file) \Rightarrow \neg assembly(file)$$

Encrypted data is an unintelligible form called cipher [14]. If a file is encrypted then it is unintelligible and hence cannot be correctly disassembled. That is, an encrypted file must be processed (decrypted) prior to, or as part of, execution. It is possible that an encrypted file will disassemble into something syntactically acceptable but semantically meaningless. This property also applies to data files. Indeed, the file might contain data which would halt the processor.

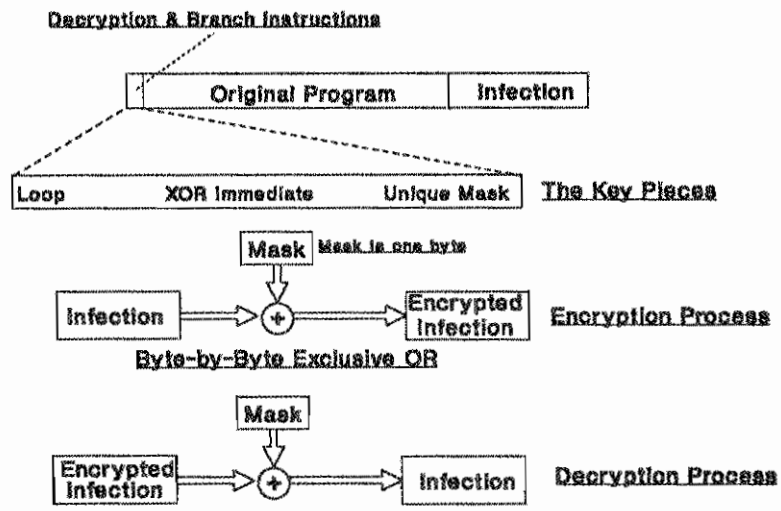

Figure 1: A Self Encrypting Virus

Rule 8: An encrypted file can be disassembled after applying Rule 5.

$$encrypted(file) \wedge applied\_to(first\_instruction, file) \wedge disassemble(file) \Rightarrow assembly(file)$$

The function performed in Rule 5 decrypts the cipher thereby restoring the code to a machine readable format. The resulting machine readable code can then be disassembled by applying Rule 6.

```
Rule 9: A known and unencrypted virus can be
        located if it resides in an executable.
```

```
¬encrypted(pattern) ∧ known_as(pattern, name) ⇒ virus(name)
```

A simple pattern matching function is sufficient to satisfy Rule 9. Many of the existing virus detecting programs search files looking for patterns representing viruses. The function "known_as" is a table look up of known viruses.

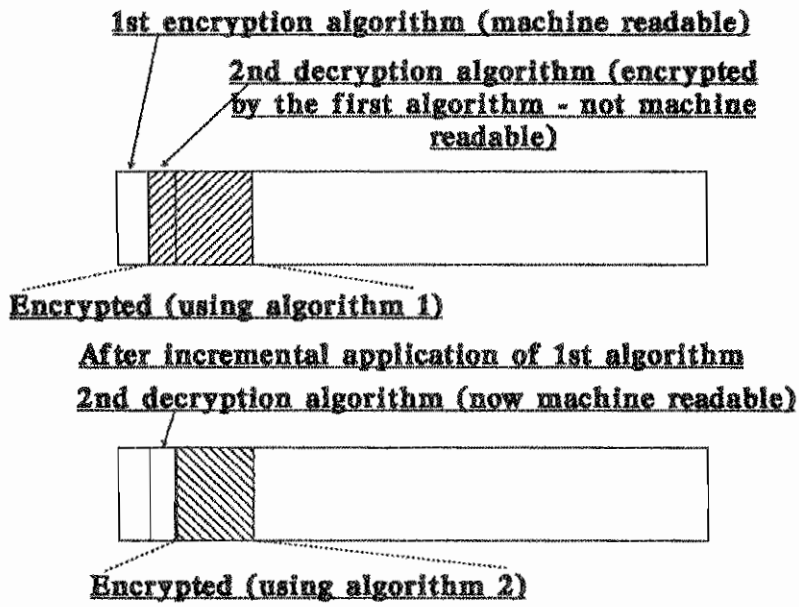

Figure 2: Double Encrypted Virus

```
Rule 10: If a binary difference from DIFF
         disassembles, the likelihood of a
         random error is low.
```

```
assemble(file) ⇒ file(executable)
```

It is remotely possible to disassemble a random file and get legitimate code, however there are sufficient invalid states to make this unlikely. Rule 10 points out that in a random corruption of a file, the probability of the corrupted difference being executable is low. Within a given CPU instruction set there are many illegal states. A random corruption would most likely result in many non-valid instructions, any of which would result in an error state.

Figure 1 and 2 illustrate a stealth encryption virus. In this simple example, the circle represents a process performing the exclusive-OR function of a MASK byte to each byte of the infection. In a more sophisticated encryption scheme, the MASK could be obtained from a key stream such that each byte-wise XOR would be with a psuedo-randomly derived MASK. The random outputs would be exclusive OR'd to each byte resulting in a stronger encryption scheme.

Software Development Rules

In a development environment changes are made to source code and then recompiled. Thus legitimate changes to executable code should follow changes to source code. If the development tools and source code remain unchanged while the executable changes, then the changed executable is probably not legitimate as shown in Figure 3.

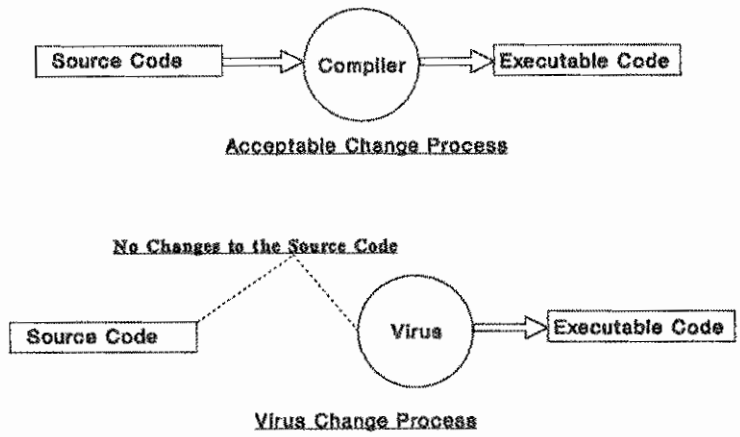

Figure 3: Development Versus Infection

```
Rule 11: If an executable program changes but the source
         code does not then the changed executable is
         probably not legitimate
```

$$configuration(unchanged) \wedge integrity(source,file) \wedge \neg integrity(executable,file) \Rightarrow \neg legitimate(executable,file)$$

If nothing changes, then compiling the same source code should result in identical executables.

Rule 12: Compiling revised source code produces revised executable code.

$\neg integrity(source,file) \wedge compile(source,file) \Rightarrow \neg integrity(executable,file)$

An interesting point made by Page [15] is the possibility of source code viruses. Given the C compiler Trojan horse example described by Thompson [21], it is not unreasonable to visualize a source code virus. To see how a source code virus might work, consider the following. By infecting only source code, it would be difficult for many of the current "executable" detectors to monitor systems. Optimizing compilers often restructure code such that the executable files might not have a discernable signature. A source infector would

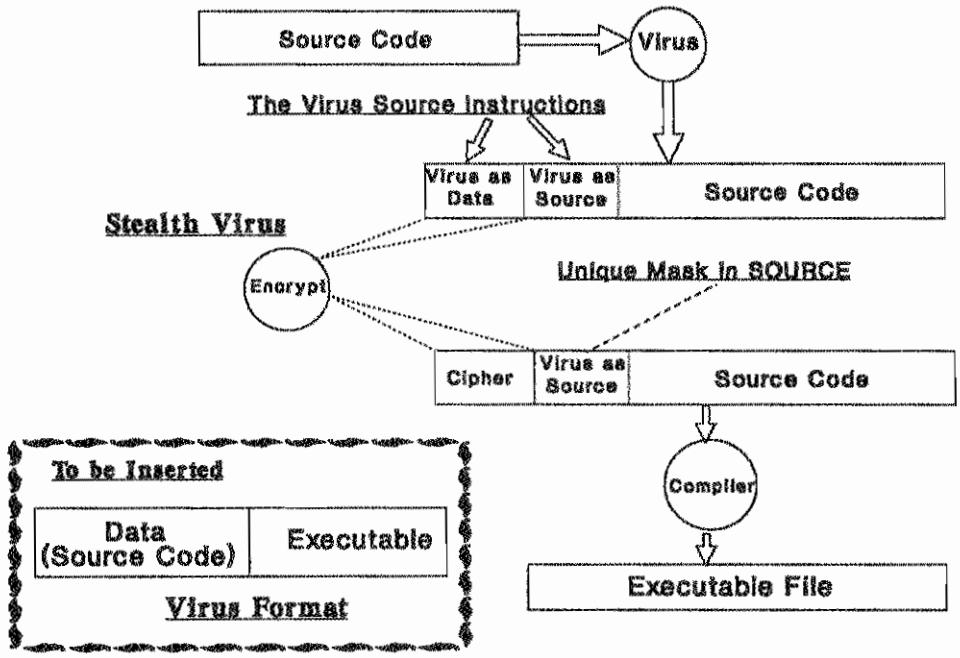

Figure 4: Source Code Virus

require several pieces including source code readable by a translator (compiler). The actual infection could insert two copies of itself into the source code. The first copy might be declared as a text array. The second copy would be destined for in-line insertion thereby becoming executable after compilation. Further, a stealth virus might encrypt the text array making executable pattern recognition more difficult as shown in Figure 4. A source code virus might be detected by comparing infected source code files to reveal identical in-line instructions representing the virus.

A typical source code infector might look like the virus format shown in Figure 4. In this example, the virus contains executable code and a text buffer containing compiler readable source code. Everything is self contained within the infection. After compilation, the resulting file contains both source and executable code. Clearly, a source code virus is feasible.

Applying the Rules

By applying the above stated rules a disassembled copy of the viral infection can be extracted. This section describes the procedure and then address what can be learned from the virus code. The specific rule addressed will be abbreviated. For example, Rule 1 will be denoted (R1).

Cohen has shown that in general, it is undecidable whether or not a sequence of code is a virus [3]. Furthermore, other researchers agree with Cohen's proof and have proposed refinements to his proof model [10]. By contrast, Crocker and Pozzo [5] proposed a "fail-safe" virus filter. Ducking the religious issues associated with these two extreme positions, there are some pieces of information which are decidable in polynomial time. For example, if we have a known virus such as nVIR we can conclude that a file is infected if we find nVIR in an executable. This example holds for the special case of a known virus, but not in general.

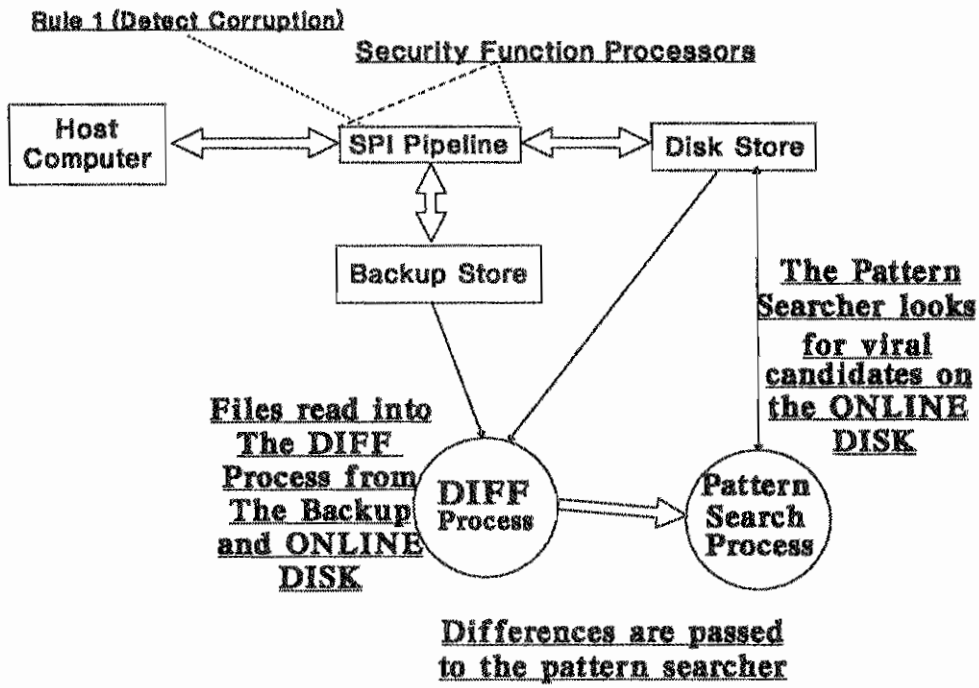

Figure 5: A Possible Virus Control

One use of assembly code is to search for illicit code as described in [8]. The assumption is that viral code has some identifiable features which differentiate it from normal instructions. A similar approach focusing on viral operating system calls was proposed in [11]. For example, in the 80x86 CPU, instructions such as IN, OUT, or INT might be cause for concern.

When a new virus hits there is time lost in figuring out what the virus does. Typical inquires request information on triggers and payloads. Much of the desired information can be obtained from a parse performed within an isolated processor. By applying a disassembly to the executable program and then searching for viral instructions, much information can be accumulated. An example environment using the SPI architecture is shown in Figure 5.

Using the SPI architecture described in [9], changes to executable files can be detected using (R2). From the altered file, the difference can be extracted by (R4). The extracted difference can be decrypted if

encrypted (R5) and then disassembled (R6). If the disassembly succeeds (R10) then there is a good indication that the file has been deliberately modified. If the unencrypted difference (R5) appears in multiple files then it is likely that a virus is at work.

Figure 5 also illustrates a possible mitigative control based on transparently restoring corrupted files from a backup disk. In this example, the backup store files are used for comparisons and are not executed on the SPI processor. Therefore, an infected file residing on the backup store could not infect the SPI processor. As a final point, the backup store could be updated using an approach similar to [13] where the user is queried.

Derivable Cases

Consider the shrink-wrap virus case. The virus would first manifest itself by executing its payload or by reproducing. If the virus is still in the reproductive stage, then it will most likely be detected in another file after that file's integrity is altered due to infection. Through the application of the rules previously stated, the newly infected file will provide a source for extracting the viral code. A pattern matcher can then explore all executable files for an occurrence of the same viral set. Should the pattern be found in an original distributed file then we can infer that the source is a shrink-wrap virus.

Summary

This paper examines inference rules involved in identifying a computer virus. The newer self encrypting stealth viruses are examined and along with the necessary conditions for their detection. The rules are then used to derive properties of new viruses.

Acknowledgements

I would like to thank Cheryl Ledbetter, Lee Rice, and the reviewers for their objective comments and recommendations.

References:

[1] Leonard M. Adleman, "An Abstract Theory of Computer Viruses", *Lecture Notes in Computer Science* Vol. 403, *Advances in Computing - Crypto '88*, S. Goldwasser (ed.), Springer-Verlag, 1990.

[2] Fred Cohen, "Computer Viruses", *Proceedings of the 7th DOD/NBS Computer Security Conference*, pp. 240-263.

[3] Fred Cohen, "Computer Viruses", pp. 23-27, Copyright 1985 by Fred Cohen.

[4] Fred Cohen, "A Cryptographic Checksum for Integrity Protection", *IFIP Computers and Security* 6(6), 1987.

[5] Steve Crocker & Maria Pozzo, "A Proposal for a Verification Virus Filter", *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy*, pp. 319-324.

[6] Russell Davis, "Exploring Computer Viruses", *Proceedings of the Fourth Aerospace Computer Security Applications Conference*, pp. 7-11.

[7] Russell Davis, "Uncovering Viruses", *Proceedings: Fourth Annual Computer Virus & Security Conference*, pp. 796-803, March 14-15, 1991.

[8] Garnett, "Selective Disassembly: A First Step Towards Developing a Virus Filter", *Proceedings of the Fourth Aerospace Computer Security Applications Conference*, pp. 2-6.

[9] Lance J. Hoffman, et al., "Security Pipeline Interface (SPI)", *Proceedings of the Sixth Annual Computer Security Applications Conference*, December, 1990.

[10] Kimmo Kauranen, et. al., "A Note on Cohen's Formal Model for Computer Viruses", *Special Interest Group - Security, Audit & Control Review*, Volume 8, Number 2, pp. 40-43, ACM Press.

[11] Paul Kerchen, et al., "Static Analysis Virus Detection Tools For UNIX Systems", *Proceedings of the 13th National Computer Security Conference*, pp. 350-365.

[12] Teresa F. Lunt, "Automated Audit Trail Analysis and Intrusion Detection: A Survey", *Proceedings of the 11th National Computer Security Conference*, October 1988.

[13] James Molini and Chris Ruhl, "The Virus Intervention and Control Experiment", *Proceedings of the 13th National Computer Security Conference*, pp. 366-373.
[14] "The Data Encryption Standard", *FIPS PUB 46*, National Bureau of Standards (Now NIST).
[15] John Page, "An Assured Pipeline Integrity Scheme for Virus Protection", *Proceedings of the 12th National Computer Security Conference*, pp. 378-388.
[16] Charles P. Pfleeger, "Security in Computing", copyright 1989 by Printice-Hall, Inc., pp. 160-161.
[17] Pozzo and Gray, "An Approach to Containing Computer Viruses", *IFIP Computers and Security*, 6(4), 1987.
[18] Michael M. Sebrint et. al., "Expert Systems in Intrusion Detection: A Case Study", *Proceedings of the 11th National Computer Security Conference*, October 1988.
[19] Eugene H. Spafford, et al., "Computer Viruses: Dealing With Electronic Vandalism and Programmed Threats", *ADAPSO*.
[20] Andrew S. Tanenbaum, "Computer Networks", Copyright 1981 by Prentice Hall, pp. 128-132.
[21] Ken Thompson, "Reflections on Trusting Trust", *Communications of the ACM*, Vol. 27, No. 8.
[22] Steve R. White, et al., "Coping with Computer Viruses and Related Problems", copyright 1989 by International Business Machines Corporation.
[23] David R. Wichers, et. al., "PACL's: An Access Control List Approach to Anti-Viral Security", *Proceedings of the 13th National Computer Security Conference*, pp. 340-349.
[24] Catherine L. Young, "Taxonomy of Computer Virus Defense Mechanisms", *Proceedings of the 10th National Computer Security Conference*, pp. 220-225.