

AD-A246 083



NAVAL POSTGRADUATE SCHOOL
Monterey, California

2

DTIC
ELECTE
FEB 20 1992
S D D



THESIS

SOFTWARE AND THE VIRUS THREAT:
PROVIDING AUTHENTICITY IN DISTRIBUTION

by

LT George M. LaVenture, USN

March 1991

Thesis Advisor: Dr. Norman F. Schneidewind

Approved for public release; distribution is unlimited.

92-03881



92 2 14 020

REPORT DOCUMENTATION PAGE				
1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			Approved for public release; distribution is unlimited.	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)			5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b. OFFICE SYMBOL (If applicable) 55		7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000			7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)			10. SOURCE OF FUNDING NUMBERS	
			Program Element No.	Project No.
			Task No.	Work Unit Accession Number
11. TITLE (Include Security Classification) SOFTWARE AND THE VIRUS THREAT: PROVIDING AUTHENTICITY IN DISTRIBUTION				
12. PERSONAL AUTHOR(S) LAVENTURE, GEORGE M.				
13a. TYPE OF REPORT Master's Thesis		13b. TIME COVERED From To		14. DATE OF REPORT (year, month, day) March 1991
				15. PAGE COUNT 82
16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
17. COSATI CODES			18. SUBJECT TERMS (continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUBGROUP		
			Computer Viruses, Virus Prevention,	
			Computer Security, Digital Signatures	
19. ABSTRACT (continue on reverse if necessary and identify by block number)				
Computer viruses have threatened the integrity and reliability of computer systems since 1983. Literally hundreds of viruses exist for the IBM compatible computer alone. These viruses can cause corruption or loss of program and data files, incidental damage to hardware, and degradation or loss of system performance. This paper examines the nature of the virus threat by discussing virus types, methods and rates of propagation, relative frequencies of occurrence, and genealogy. Possible methods for virus detection and identification, followed by disinfection, are outlined. Minimum capabilities and testing criteria for these products are also detailed. Methods for controlling and limiting infection and damage are discussed. These are considered minimum acceptable safeguards to be implemented by an organization. Lastly, software authentication means are examined, which, when used in conjunction with the minimum safeguards, would eliminate the possibility of viral infection.				
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT XX ☒ UNCLASSIFIED/DUNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Norman P. Schneidewind			22b. TELEPHONE (Include Area code) (408) 646-2719	
			22c. OFFICE SYMBOL AS-SS	

Approved for public release; distribution is unlimited.

Software and the Virus Threat:
Providing Authenticity In Distribution

by

George M. LaVenture
Lieutenant, United States Navy
B.S., 1981 Syracuse University

Submitted in partial fulfillment
of the requirements for the degree of

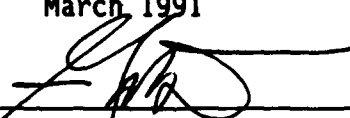
MASTER OF SCIENCE IN COMPUTER SYSTEMS MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL

March 1991

Author:



LT George M. LaVenture, USN

Approved by:



Dr. Norman F. Schneidewind
Thesis Advisor



Mr. John W. Mildner
Second Reader



Dr. David R. Whipple, Jr.
Chairman, Department of Administrative Science

ABSTRACT

Computer viruses have threatened the integrity and reliability of computer systems since 1983. Literally hundreds of viruses exist for the IBM compatible computer alone. These viruses can cause corruption or loss of program and data files, incidental damage to hardware, and degradation or loss of system performance.

This paper examines the nature of the virus threat by discussing virus types, methods and rates of propagation, relative frequencies of occurrence, and genealogy.

Possible methods for virus detection and identification, followed by disinfection, are outlined. Minimum capabilities and testing criteria for these products are also detailed.

Methods for controlling and limiting infection and damage are discussed. These are considered minimum acceptable safeguards to be implemented by an organization.

Lastly, software authentication means are examined, which, when used in conjunction with the minimum safeguards, would eliminate the possibility of viral infection.

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	BACKGROUND	1
B.	SOFTWARE CATEGORIES	2
C.	MICROCOMPUTER RELIABILITY	3
II.	NATURE OF THE VIRUS THREAT	5
A.	NAME ORIGIN	5
B.	TYPES OF VIRUSES	5
1.	Boot Infectors	6
2.	System Infectors	7
3.	Executable Program Infectors	7
C.	PROPAGATION ESTIMATES	9
D.	RELATIVE FREQUENCY	11
E.	VIRAL GENEALOGY	13
III.	VIRUS IDENTIFICATION AND REMOVAL	15
A.	IDENTIFICATION	15
1.	Infection Preventors	16
2.	Infection Detectors	17
3.	Infection Identifiers	18
B.	REMOVAL	19
1.	Virus Specific Disinfectors	20

2. Universal Disinfectors	20
IV. VIRUS INFECTION PREVENTION METHODS	24
A. USER TRAINING	24
1. Basic Precautions	24
2. Virus Recognition	25
B. HARDWARE MEASURES	26
1. Write Protect Tabs	26
2. Tamper-proof Shrinkwrap	27
3. CD ROM	27
C. SOFTWARE MEASURES	28
1. Virus Scanners	28
2. Authentication Methods	28
V. AUTHENTICATION METHODS	31
A. CHECKSUMS	32
B. CYCLIC REDUNDANCY CODES	32
C. ENCRYPTION	34
1. Cryptographic Systems	36
2. Reasons for Cryptography	37
a. Secrecy	37
b. Authenticity	37
3. Types of Cryptosystems	38
a. Symmetric Cryptosystems	38
b. Asymmetric Cryptosystems	38
D. MESSAGE AUTHENTICATION CODES	39

1. Public Key Cryptosystems	39
a. Providing Secrecy.	40
b. Providing Authentication	40
c. Secrecy with Authentication	40
2. Message Digests	41
a. Hash Functions	41
b. The RSA Signature Scheme	42
E. HYBRID SYSTEMS	43
VI. PRACTICAL SOFTWARE AUTHENTICATION	45
A. MD4	45
B. SIGN AND CHECK	46
C. VIRUS-SAFE	46
VII. CONCLUSIONS AND RECOMMENDATIONS	48
A. CONFIDENCE BUILDING	48
1. User Training	49
2. Virus Detection and Removal	49
B. ASSURANCE BUILDING	50
1. Software Authentication	50
C. THE BOTTOM LINE	52
VIII. APPENDICES	54
A. EXAMPLES OF DOS VIRAL INFECTION IN THE US	54
B. CHRONOLOGY OF A VIRUS AS TOLD BY IT'S AUTHOR	55
C. KNOWN VIRUS INFECTION AND DAMAGE CHARACTERISTICS	56

D.	ANTI-VIRAL PRODUCT MINIMUM CAPABILITIES LIST	60
1.	Infection Prevention Products	60
2.	Infection Detection Products	60
3.	Infection Identification Products	61
E.	ANTI-VIRAL PRODUCT EVALUATION PROCEDURES	62
1.	Infection Prevention Products	62
2.	Infection Detection Products	63
3.	Infection Identification Products	63
F.	ANTI-VIRAL SOFTWARE	65
1.	Infection Prevention Systems	65
2.	Infection Detection Systems	65
3.	Infection Identification Systems	65
G.	MD4 LISTING	66
IX.	LIST OF REFERENCES	71
	INITIAL DISTRIBUTION LIST	73

ACKNOWLEDGEMENT

I would like to give special thanks to the following individuals who spent their time discussing the subject with me.

Mr. Mike McLaughlin	Navy Information Resources Management
Ms. Judith Froscher	Naval Research Laboratory
Mr. Bruce Calkins	National Computer Security Center
Mr. Bruce Coster	National Computer Security Center
Dr. Dennis Branstad	National Institute of Science and Technology
Mr. John Wick	National Institute of Science and Technology

Thanks to Mr. Kenneth Van Wyk, moderator of the VIRUS DISCUSSION LIST, for sending me his excellent product.

Very special thanks to my advisor, Dr. Norman F. Schneidewind, at the Naval Postgraduate School, Monterey, CA, and my second reader, Mr. John Mildner, at the Naval Electronic Systems Security Engineering Center, Washington, DC, for their patience and efforts to help me complete this research project.

This research was supported, in part, by the Naval Electronic Systems Security Engineering Center, Washington, DC.

I. INTRODUCTION

A. BACKGROUND

Since the first infectious and destructive computer virus was created in November 1983¹, and the first microcomputer virus in January 1986², the computer security field has never been the same.³ Computer viruses have received wide reporting in both trade journals and the general press. Viral code written in Asia could be "exported", via modem or mail, around the world.⁴ Systems could be infected quicker than warnings could be received and precautions taken.⁵ The recent, and much publicized, UNIX Worm and AIDS Trojan incidents are but two examples of the damage malicious code can do.

¹ While conducting Doctoral Thesis research at the University of Southern California in 1983 and 1984, Fred Cohen developed the first computer virus and conducted propagation experiments on a VAX computer with a UNIX operating system.

² The virus, later named Pakistani Brain, originated in Lahore, Pakistan. It was developed by two brothers purportedly as a copy protection scheme for software they sold in their store. The original version of this virus has their names and telephone number programmed in the code.

³ For comparison, the IBM PC was announced in 1980 and went on sale October 1981.

⁴ The Pakistani Brain spread rapidly to North America via Europe. In less than twelve months it had infected nearly a half-million computers in hundreds of universities, corporations and government agencies.

⁵ Cohen conducted five trial runs in which his virus never took more than an hour to infect the VAX system. The shortest time to full infection was five minutes, the average half an hour. His work was so successful that university officials refused to allow further experiments.

The Department of the Navy (DON), in its drive toward technological sophistication, is becoming increasingly computer dependant. Ships are receiving administrative microcomputers and "smart" weapons systems while shore establishments have their management information systems interconnected by wide and local area networks (WANS and LANS). This dependency and interconnection increases the potential of viral infection and the threat of data compromise and degradation or loss of system performance. Regardless of the source, a campus prankster or a foreign power, protecting our systems from viruses will be essential to ensure their reliability.

B. SOFTWARE CATEGORIES

Software used by DOD can be broadly categorized as either mission critical or mission support. Mission critical software directly impacts on DOD's ability to defend the United States from attack. Such software would include missile guidance systems and military forces command and control programs. Mission support software, all other DOD software not directly effecting the defense of the United States, would include payroll packages, personnel databases, and office automation.

Development of mission critical software often requires access to classified hardware design and performance specifications. Depending upon classification, special storage and development facilities, access procedures, and testing criteria may be employed. Additionally, the fielding of hardware and software systems would most probably be performed through a secure distribution channel.

Mission support software development will, in general, require access to unclassified, or at most, unclassified but sensitive data¹. Many of the restrictions for classified projects may not apply. Distribution of hardware and software will be through the Central Design Agent (CDA), the standard supply system or via open purchase. Unfortunately, this increases the vulnerability of our systems since the relative percentage of word processors procured by the Navy exceeds the number of missile guidance programs.

Due to the comparatively open nature of development, the relative percentages of procurement, and the underlying simplicity of the custody chain, I chose to examine viral protection of mission support software.

C. MICROCOMPUTER RELIABILITY

The IBM compatible microcomputer's popularity, widespread availability, and general lack of security has made it the target of most viral attacks in the last five years. The threat has become so wide-spread that Allstate Insurance Company now offers virus insurance. Its home and business insurance policies have been extended to cover viral damage to microcomputers. (Skulason, 1990, #3-35) These are the same systems which have been used aggressively for office automation, command LANs, and access to sensitive command and control systems such as the Worldwide Military Command and Control System (WWMCCS).

¹ The Computer Security Act of 1987, signed into law 8 January 1988, created this category of information. It includes privacy act and contract sensitive data.

With this in mind, I will focus on mission support software for IBM compatible hardware only.¹ Providing viral safeguards for these systems is a first step toward overall computer system protection. The question then, is "How do we provide protection from viral attack?".

I will broadly define a virus as any program which replicates and spreads itself secretly. Assuming a given computer is not infected when manufactured², the infection must occur during use. This implies the infection vector³ is the software which is then added to it by the user. We can then narrow our research question to "How do we prevent the loading of infected software?".

Before answering, we must understand the nature of the threat, the types of existing viral detection and removal tools, and potential means of protecting software. These issues will be addressed in the following chapters.

¹ This is not unrealistic since the vast majority of microcomputers dedicated to mission support functions are IBM compatible. Indeed many competitively bid procurement contracts have specified this compatibility as a requirement.

² A valid assumption since memory is empty and any disk drives are empty or unformatted.

³ "An agent capable of transmitting a pathogen from one organism to another either mechanically as carrier or biologically by playing a specific role in the life cycle of the pathogen." [Webster's Third New International Dictionary]

⁴ Commercial, shareware, or public domain only, since I assume a software developer will not write code to deliberately infect and damage his own system.

II. NATURE OF THE VIRUS THREAT

A. NAME ORIGIN

Virus is a normal Latin 2nd declension word meaning 'slime', 'poison', and 'offensive'. While its first English usage was in 1599, it was not used in its present meaning as 'filterable virus'¹ until 1880. [Oxford English Dictionary, Second Edition] The invisible and destructive nature of the biological virus led to its adoption as the name for its electronic cousin. In fact, many researchers use terms reminiscent of the biological virus: vector, infection rate, and vaccine. The plural of 'virus' as used in English, is 'viruses'.²

B. TYPES OF VIRUSES

Computer viruses can be categorized in three major classes based upon their area of system residence and/or infection:

- boot infectors
- system infectors
- executable program infectors

¹ "An infectious organism, usually submicroscopic, that can multiply inside certain living host cells. A non-cellular structure lacking any intrinsic metabolism usually comprising a DNA or RNA core inside a protein coating." [Oxford English Dictionary, Second Edition] It would pass through filters that would stop bacteria.

² Of note is the significant disagreement between academicians concerning the 'true' plural of 'virus'. The first quarter of 1990 saw weeks of electronic word war via the VIRUS DISCUSSION LIST and other research oriented electronic forums concerning this point. For my part, I use 'viruses' throughout this work to represent the plural.

1. Boot Infectors

These viruses reside in a disk's¹ boot sector.² If active in memory, a boot virus will infect a new disk by relocating the boot sector contents to a previously empty disk sector and marking it as bad in the File Allocation Table (FAT).³ The virus then adds a jump instruction to its end and writes a copy of itself to the boot sector.⁴ The jump ensures that, after the boot virus is loaded into memory and executed during booting, computer control is passed to the original boot code at its new location.

An infected disk can infect the system whenever the disk boot sector is executed.⁵ While memory resident, these viruses can infect any

¹ DOS disks are organized using a rigid scheme. Each disk in a drive is divided into one or more logical volumes. Each logical volume consists of four areas: the boot sector containing configuration and bootstrap information, an original and backup File Allocation Table (FAT) which holds cluster chaining and ownership information, the disk root directory which holds information pointing to the first cluster in the FAT chain holding a given file's or subdirectory's data, and the file area which consists of clusters maintaining file data chained by the FAT pointers.

² The boot sector, logical sector contains critical information regarding the disk medium such as: DOS name and version, bytes/sector, sectors/cluster, number of reserved sectors, number of FATs, number of root directory entries, total sectors in the logical volume, media descriptor byte, number of sectors/track, number of disk drive heads, number of hidden sectors, and the disk bootstrap to load the operating system from disk (the ROM bootstrap is smart enough to home the disk drive head, read the boot sector from disk, and jump to it in memory).

³ The "bad" sector marking in the disk FAT ensures that these sectors will not normally be examined or altered by the system.

⁴ Boot infectors typically mark several "good" disk sectors as "bad". These sectors are then used to hold the original boot sector code plus whatever virus code would not fit in the boot sector.

⁵ A disk's boot sector is examined whenever drive hardware detects a diskette change or upon system reset for logical drive 0 only.

disk in the system. These viruses are fairly tame since they infect a given disk only once and are relatively easy to find.

2. System Infectors

These viruses attach themselves to the command interpreter and other system files that remain memory resident and reside on bootable hard or floppy disks.¹ Except for exclusively targeting system files, these viruses behave similarly to executable program infectors which are discussed below.

While the relatively small number of systems programs should make these viruses somewhat tame, the fact that systems programs remain memory resident and are frequently called allow these viruses to cause a high degree of infection in a short span of time.

3. Executable Program Infectors

These viruses are particularly troublesome since they can spread to any executable program² in the system by either appending or overwriting. A virus generally appends itself to either the front or back end of an executable file.³ Front end appenders situate their code so it is

¹ The 8086/8088 family of microprocessors are designed so that, when reset or powered up, program execution begins at memory address 0FFFF0H. This lies within ROM memory and contains a jump instruction to the system power up self test (POST) and bootstrap code. The bootstrap code loads and executes the system programs MSDOS.SYS and IO.SYS. IO.SYS ultimately loads and executes the command interpreter COMMAND.COM.

² Any program ending with the suffix COM, EXE, OVL, or BIN is considered by the operating system to be executable.

³ According to John McAfee, Chairman of the Computer Virus Industry Association (CVIA) and President of McAfee Associates, a Santa Clara, California based anti-viral research and marketing firm:

executed before the host program. After the virus performs its task, it then returns control to the legitimate code. Back end appenders usually add a JUMP instruction in the front end pointing to the viral code. After virus execution, another JUMP points back to the original program code. Overwriting viruses simply replace a section of the existing code with their own instructions. This subgroup is usually detectable earlier in the infection process since the host program may no longer function correctly. The appenders may slow program performance but will generally escape detection until the virus damage sequence is triggered.

This type virus usually accomplishes its infection by either:

- copying itself to another executable file whenever an infected program is executed and then passing control to the host program
- by remaining memory resident and infecting each program that is loaded into memory

During infection, the original file size, date, and time may be changed. However, sophisticated viruses may save and restore the original values when writing the modification to disk. Additionally, to avoid early detection and maximize infection, the virus may avoid previously infected

"Viruses can attach to a program's beginning, end, middle, or any combination of the three. They may fragment and scatter virus segments throughout the program or keep the main body of the virus unattached to the program, hidden in a bad sector. All known viruses, however, [modify the program's beginning to] ensure the virus is executed before the host. If this were not so, the uncertain environment in which the virus executed would increase the possibility of program failure [and early detection]. Viruses which replace entire programs, such as boot infectors, and viruses that attack only specific programs [such as system infectors], are the only exception to this rule. These viruses may gain control at any point, since the structure of the host program is well known and the environment can be predicted." (McAfee, 1989)

files or delay its damage sequence until infection has reached a predetermined level.

C. PROPAGATION ESTIMATES

Estimates of viral multiplication rates are not easily obtained for many reasons:

- computer hardware may remain constant but software used and preventative measures taken may vary greatly from site to site and machine to machine
- many researchers are reluctant to divulge their estimations since they are often derived from reports concerning products they are supporting
- this information is still considered 'embarrassing' and 'sensitive'

Dr. Fridrik Skulason, virus researcher at the University of Iceland, Technical Editor of the Virus Bulletin (UK), and consultant to the Naval Computer Incident Response Team (NAVCIRT) at the Naval Electronic Systems Security Engineering Center (NAVELEXSECCEN) in Washington, DC, has, however, recently released estimates for two of the oldest and most wide-spread viruses.¹ These appear in Figure 1.

Total number of PCs	30.000.000 machines
Number infected with Jerusalem	100.000-500.000 machines
Number infected with Brain	100.000-500.000 machines

Figure 1 - Estimated PCs Infected with Jerusalem and Brain (Skulason, 1990, #3-64)

¹ The Jerusalem and Pakistani Brain viruses are about 51 and 74 months old, respectively.

Figure 2 provides the amended estimate if each infection¹ on the same machine is counted.

Number of Jerusalem infections	2.000.000-10.000.000 infections
Number of Brain infections	1.000.000- 5.000.000 infections

Figure 2 - Estimated Jerusalem and Brain Infections (Skulason, 1990, #3-64)

Of note is the apparent virility of Jerusalem compared with Brain even though Brain is a third older. This is primarily due to its targeting of executable programs instead of the comparatively rare disk boot sectors.²

Skulason hypothesizes that viral infections increase exponentially over time but slow as the virus saturates the system. This can be seen in Figure 3. His experience with organizational infections³ indicates that once a virus infects a computer, it will usually spread organization wide in one to two months. (Skulason, 1990, #3-64).

¹ Skulason estimates that 20 infected programs reside on every Jerusalem infected machine, and 10 diskettes have been infected by every Brain infected computer.

² According to John Mildner, head of the Naval Computer Incident Response Team (NAVCIRT) at the Naval Electronic Systems Security Engineering Center (NAVELEXSECCEN) in Washington, DC: "Jerusalem probably spreads more rapidly since it uses executable files as the infection vector. These files are often transferred electronically via bulletin boards or computer networks. On the other hand, the virus most common to the Navy, the Stoned boot sector virus, is spread by exchange of data files on floppy diskette." (Mildner, 1991)

³ Organizations in Iceland are not that large - The Bank of Iceland is one of the largest and had about 700 PCs as of mid 1990.

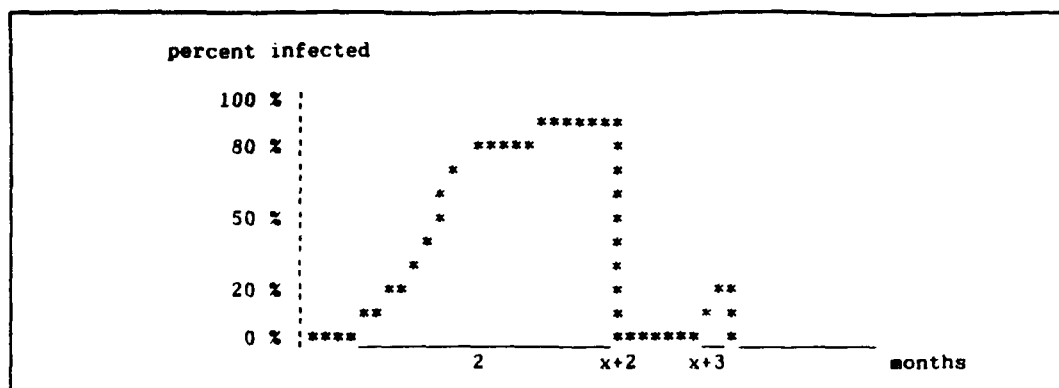


Figure 3 - Percentage of Organization Infected vs Time (Skulason, 1990B)

Skulason shows that the number of infected computers will rise slowly when the virus is first introduced, but, assuming favorable conditions¹, a virus will infect 80% of the machines within 2 months. The virus will then probably remain unnoticed for some time.² Once detected, it is usually removed swiftly, but, almost never completely.³ When it reappears a month or two later, some preventive software is finally installed and the virus is defeated.⁴ (Skulason, 1990, #3-64)

D. RELATIVE FREQUENCY

A virus's frequency of occurrence is related to its propagation rate and age. Viruses which propagate faster such as executable infectors will

¹ Such as lack of preventive hardware or software, or significant disk or program traffic between different computers.

² This is the period in which the virus actively replicates without any noticeable performance degradation or damage sequence.

³ Particularly if backup files loaded to recover lost data files have been corrupted.

⁴ Skulason estimates that, in Iceland, about 5% of all PCs have been infected with a virus. However, viewing single organizations, he either finds no viruses or an 80% infection rate.

be seen more frequently than slower propagators like boot infectors. Likewise, older viruses have had longer to establish themselves in the user community and will be spotted more frequently than viruses which have just been isolated. Figure 4 lists the relative frequency of viruses as observed by David Chess, Fridrik Skulason, and Morton Swimmer.

	USA	Iceland	PRG
	---	-----	---
Bouncing Ball	26%	30 %	10%
1813 (Jerusalem)	21%	5 %	15%
1704	15%	50 %	20%
Stoned	9%	2 %	10%
1701	8%	5 %	5%
648 (Vienna)	7%		10%
Brain	7%	2 %	
Yale	1%		
17Y4	< 1%		
2772 (Y.D.)	< 1%		
765	< 1%		
Disk Killer	< 1%	2 %	5%
Lehigh 1	< 1%		
Sunday	< 1%		
Sylvia	< 1%		
Icelandic 1/2/3		3 %	
Ghostballs		1 %	
Macho			1%
Advent			< 1%
Dark Avenger			2%
5120			< 1%
Vacsina (TP04)			5%

Figure 4 - Observation Frequencies (Chess, 1990)(Skulason, 1990, #3-91)(Swimmer, 1990)

These figures indicate that at least 70 percent of the infections are caused by half a dozen viruses. As of February 1991, 222 major virus strains have been isolated. Therefore, more than 70 percent of infections are caused by less than 3 percent of the known strains.¹ This makes the virus identification and removal problem much more manageable.

¹ John McAfee identifies 10 viruses (Pakistani Brain, Jerusalem, Alameda, Cascade, Ping Pong, Stoned, Lehigh, Den Zuk, Datacrime, and Fu Manchu) which, he believes, represent 95 percent of reported infections.

Details of specific infection cases in the United States are provided in Appendix A (McDonald, 1990). A chronology of the Dark Avenger virus, as told by its author, can be found in Appendix B (Skulason, 1990, #3-97).

E. VIRAL GENEALOGY

As the number of viruses increase, so does the difficulty of tracking their inter-relations. Figures 5 and 6 provide the genealogy for those viruses considered related to others.

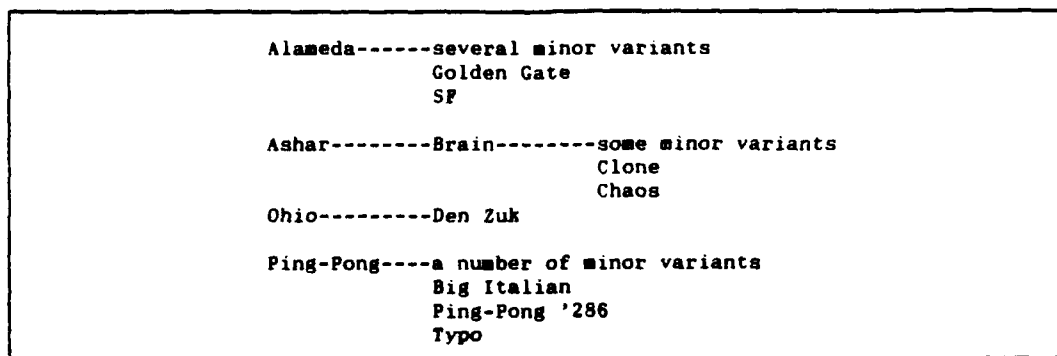


Figure 5 - Boot Infectors (Skulason, 1990, #3-89)

Like its biological counterpart, a computer virus tends to evolve and mutate. This mutation is controlled by programmers who generate new viruses from the source or disassembled code of older ones. This process often precludes major changes or enhancements and may assist identifying new viruses soon after they appear.¹ Appendix C (McAfee, 1991) details infection and damage characteristics for known IBM compatible viruses.²

¹ A recent development, however, has been the use of self encryption to hide and avoid detection.

² As of 26 February 1991.

```

April 1st COM----+
      +----Surv 3.0---Jerusalem---many minor variants
April 1st EXE----+
                        Centurv
                        Fu Manchu
                        Sundav
                        Payday
                        PSQR
                        Prudens
                        Anarkia

Cascade (1701)---1704-----many minor variants
                        1704 Format
                        17Y4

Dark Avenger----Dark Avenger 2000

Datacrime 1168----Datacrime 1280----Datacrime 2----Datacrime 2B

Icelandic----+----Saratog.
      +----Mixl---      -Mixl B
      +----Iceland. 2-----December 24th

Old Yankee-----+
                        +----Yankee (around 15 variants)
Vacsina (over 20 variants)----+

Pixel-----many Bulgarian variants
      Amstrad

South African---+----a few minor variants
      +----Virus-B---some minor variants

Syslock---+----Macho
      +----Advent

Traceback-----2930

Vienna---+----Burger's variant----Lisbon
      |                                1260
      |                                Ghostballs
      +----New (Bulgarian) Vienna-----many minor variants

Virus-90-----Virus-101

```

Figure 6 - Executable Program Infectors (Skulason, 1990, #3-89)

III. VIRUS IDENTIFICATION AND REMOVAL

A. IDENTIFICATION

Virus identification can either be inadvertent or deliberate. For the average PC user, viruses are unknown or something that infects 'others'. Either way, little or nothing is done to catch infection before the damage sequence is initiated. Infection is inadvertently discovered when the virus triggers, performs its destruction, and possibly identifies itself.

Deliberate identification involves the active use of various software tools tailored to locate, identify, and, perhaps, remove viruses. These tools fall into three major categories:

- programs which prevent infection
- programs which detect infection after it has occurred
- programs which identify pre-existing infection

John McAfee, Chairman of the Computer Virus Industry Association (CVIA)¹ and President of McAfee Associates², a Santa Clara, California based anti-viral research and marketing firm, points out that:

"These products, however, are not always clearly separable in the marketplace. Some combine two or more programs, each addressing a

¹ CVIA is reputed to consist of 95 percent of the anti-viral community. Most of the members are vendors of anti-viral products. It is a source of anti-viral programs and general virus information. Additionally, it is one of the few groups providing cross system (IBM, Macintosh, etc) information.

² McAfee products are discussed in this paper for illustrative purposes only. This does not represent endorsement by the author or the Department of the Navy. They represent the entire product spectrum.

single protection package, into a single package. Others may focus on a single type of protection but only provide a partial solution. For example, there exist infection detection products that will only detect changes to operating system files, ignoring all other executable code." (McAfee, 1989)

Minimum capabilities and evaluation procedures for these tools have been defined by the CVIA and are listed in Appendix D and E, respectively (McAfee, 1989). Appendix F lists some of the commonly used anti-viral products and their vendors.

1. Infection Preventors

These programs normally monitor the system watching for characteristic viral activities¹. If detected, they suspend activity, alert the user, and offer to terminate the suspect program.

The primary benefit is that all suspicious activity will be caught and viruses will not infect the system or spread.² This means that strain identification and removal is not required. Conversely, the drawbacks include numerous interruptions to valid programs³ using these system

¹ Such as absolute sector disk I/O, disk I/O to systems files, and alteration of interrupt vectors.

² With the exception of boot infectors. No software technique can prevent initial infection from a boot virus.

³ McAfee believes these programs require a user with a "fair amount of technical competence" to discriminate between legitimate program activity and a real virus threat.

"Some applications modify their executable modules during the configuration phase. Compilers, assemblers and linkage editors legitimately modify or replace executable code. The DOS SYS command will legitimately modify the boot sector and operating system files. These and other programs may cause anti-viral products to flag the activity and notify the user. The user must have sufficient knowledge of the program or activity in process to determine whether to allow it to proceed or to terminate it. Many system users do not have the necessary technical depth to make a valid decision" (McAfee, 1989)

calls, user de-sensitization due to false positives¹, and reliance on the virus being 'well behaved'².

These products should be tested, prior to use, to determine the degree of protection afforded executable files and the product's sensitivity to valid activities which might appear suspicious.

2. Infection Detectors

Detection products operate by vaccination or status logging. Vaccinators modify executable code to include a self test module which generates a run-time warning whenever the code has been modified. Status loggers create a baseline file containing key file information (sizes, dates, checksums, etc.) and routinely recheck this information to see if it has changed. If a modification is discovered, a warning message is given to indicate the areas of infection. In both methods the key assumption is that the system is not infected prior to installation.

¹ If the product flags too many legitimate activities, the user becomes conditioned to respond with 'continue' without reading the warning.

² The assumption that viruses perform I/O through system calls or software interrupts is reasonable because it allows execution on a wide variety of hardware. John McAfee, however, notes that some virus designers are:

"taking the extra effort, and running the increased risks, of interfacing directly to the hardware input/output devices. By doing so, they completely neutralize the infection prevention products' interrupt monitoring. No matter how cleverly software interrupts are trapped, or memory monitored, it is ineffectual if the virus never gives up processor control through an operating system call."
(McAfee, 1989)

Advantages include checking at boot time instead of program execution and avoidance of interrupt monitoring¹. Drawbacks include on line baseline file maintenance making it vulnerable to tampering, the necessity of strain identification and disinfection, and the assumption that the system is infection free when the baseline is generated. A serious drawback of vaccination products is that they can not detect viruses that replace code rather than modify it since any vaccination code would never have an opportunity to execute.²

Since these products identify infection after it has occurred, testing should be conducted, prior to use, to determine if they can detect modifications to executable programs such as the boot sector, the operating system, or an application program. John McAfee explains that:

"Many detection products use the virus attachment profile to speed system checking. If every byte of every program is processed in some comparison technique global checking may take some time. Systems containing many hundreds of large programs, may require anywhere from 5 to 15 minutes to complete the audit. Since a global scan should be performed at least daily, this time requirement is a significant nuisance to the average user and a deterrent for the implementation of the product. Products that only look for the characteristic initial instruction modifications, on the other hand, would complete the same audit in a matter of seconds."(McAfee, 1989)

3. Infection Identifiers

Identification products scan a system using signature byte strings to uniquely identify disk or file infection and may often provide

¹ Regardless of a virus's sophistication, some executable code will have changed after infection.

² Boot sector viruses, for example, replace the boot sector with themselves.

disinfection and recovery assistance. Their primary function, however, is identification.¹

The main benefits are the lack of assumptions and the relative speed of execution. The main drawback is the almost constant stream of new product releases² required as new viruses are isolated and old ones mutate³.

John McAfee describes the development of identification products:

"The virus must first be discovered and isolated. Then it must be disassembled and analyzed. Finally an effective countermeasure must be designed, implemented, and distributed to the public. The time lag for this process is a few months to a year or more. This window of opportunity for new virus developers will be a continuing barrier for such products." (McAfee, 1989)

These products can best be evaluated by their success at removing an actual infection.

B. REMOVAL

Fortunately for the microcomputer user community, if viral infection can be identified prior to the initiation of the damage sequence, a virus can usually be removed without loss of data or program files. Two general categories of disinfection products exist:

- products which remove a specific virus or group of viruses
- products which remove all viruses

¹ They can determine if a system is clean prior to installing a prevention or detection product.

² McAfee releases a new version of VIRUSCAN about every 3 months.

³ Viral researchers must then identify a signature byte string which uniquely identifies it. That signature string must be included in the next release of the anti-viral product.

1. Virus Specific Disinfectors

These products are able to target a specific virus strain or group of strains for removal.¹ They are often included with, or as part of, a detection or identification product.² Authors of these programs frequently release revisions as new or mutated viruses are isolated.³

As elsewhere, terminology is a point of contention within the anti-virus industry. With luck, the following terms⁴ will become industry standards:

- signature string: a sequence of bytes used by anti-viral programs to check if a program is infected
- identification string: a sequence of bytes used by a virus to check if a program is infected

2. Universal Disinfectors

A universal virus detector and disinfecter (UVD) would detect and remove viral infections both known and unknown. This, of course, depends on our ability to define viral activity under all circumstances.

We can appreciate the problem by refining our virus definition. Fred Cohen defined a virus as "a program that can infect other programs

¹ As of February 1991, 217 major virus strains exist. If modifications to these viruses are counted, there are 475 known IBM PC viruses.

² McAfee's VIRUSCAN can remove many of the most common viruses. The disinfection product CLEANUP is also provided to remove all viruses which VIRUSCAN is capable of identifying.

³ John McAfee releases a new version of CLEANUP about every 3 months.

⁴ The terms have been defined as used by IBM, but, they have been used interchangeably by other researchers.

by modifying them to include a slightly altered copy of itself" (Cohen, 1984). Unfortunately, this remains valid only if modernized as follows:

- "program" should also include boot sectors, INITs and all other forms of executable code
- "include" must also cover viruses that overwrite the victim and may destroy it completely
- "slightly altered" is inaccurate. One can imagine a virus¹ that includes only a part of itself in infected programs (Skulason, 1990, #3-24)

Consider Figure 7.

```
Program P1
  Display "This is a copy utility"
  Display "Name of input file ?"
  Input In-File
  Display "Name of output file ?"
  Input Out-File
  Copy In-File to Out-File
End
```

Figure 7 - Copy Utility Pseudo-Code (Skulason, 1990, #3-24)

¹. Skulason details a program with three parts:

- Part A contains the main program
- Part B contains program locating and memory residency procedures
- Part C contains I/O routines.

Assume P1 contains A+B and P2 contains A+C. Singularly, since they are unable to replicate, they are not viruses. When executed, P1 and P2 will not infect other programs. P1 will hide B in memory and execute the original program. P2 can check if B is present in memory. If so, A, B and C are combined in memory and executed. This new program uses B to find more programs to infect, using C, with A+B or A+C. The program A, B and C is a virus. However, it includes inert code instead of a slightly altered copy of itself in other programs. The virus only activates when its parts are combined. (Skulason, 1990, #3-24)

If we tell P1 the input file is itself and some existing program is the output file, P1 will behave, and be classified, as an overwriting virus.

P1, however, is not a virus since:

- It asks for the name of source and target files; and,
- It destroys the victim instead of executing it after the virus.

Objection 2 may not be valid, however, since this is how some existing viruses work. (Skulason, 1990, #3-24)

Consider Figure 8.

```
Program P2
  Display "Name of output file ?"
  Input Out-File
  Copy P2 to Out-File
End

Program P3
  Display "Name of input file ?"
  Input In-File
  Select Out-File at random
  Copy In-File to Out-File
End

Program P4
  Select Out-File at random
  Copy P4 to Out-File
End
```

Figure 8 - Potential Virus Pseudo-Code (Skulason, 1990, #3-24)

We want a UVD to identify P4 as a virus. It should also indicate that P2 and P3 might be virus like in some circumstances. Unfortunately,

¹ As well as most operating systems since they 'infect' other programs in the same way. A compiler used to compile a copy of itself would also qualify.

determining those circumstances is where the difficulty arises.¹ (Skulason, 1990, #3-24)

¹ Assuming that the examined program and it's environment are of finite size, I/O operations transfer finite amounts of data, and the UVD program runs on a different machine, a UVD may be possible but only on a machine many orders of magnitude more powerful than that running the examined program. If the first is a Sinclair ZX80, with 1K of memory, the second would need to be more powerful than all current super computers combined. (Skulason, 1990, #3-24)

IV. VIRUS INFECTION PREVENTION METHODS

Prevention can be divided into three areas:

- user training
- hardware measures
- software measures

A. USER TRAINING

The basic building block of security is user training. Hardware and software can not protect a user from himself. A solid foundation in anti-viral measures can be laid by a two step training program:

- basic precautions
- virus recognition

1. Basic Precautions

Most viral damage can be easily prevented by simple user precautions:

- boot only from write protected copies of the distribution diskette
- if the system has a hard disk, never boot from a floppy disk
- set file attributes to read only on as many executable files as possible to help prevent spread of infection

- consider public domain, shareware, and borrowed software infected until proven clean¹
- watch for changes in system activity such as increased disk accesses, unusual error messages, lost disk space, disk access at odd time, decreased available memory, slow response, or unusual screen displays or sounds
- in a network environment, load proven software to a file server and set file attributes to read only
- do not use master/distribution diskettes as working diskettes
- keep master/distribution diskettes in a safe place, away from the computer working area
- make frequent backups of changing data files

2. Virus Recognition

A Virus Simulation Suite is available to mimic the visual and audible effects of the most common microcomputer viruses.² Like their infectious counterpart, these programs are terminate and stay resident routines. Unlike the real virus, however, they can be turned on and off by the user and are not infectious. Some have a programmable delay, usually in minutes, to simulate the replication period. Simulations currently available include:

¹ Actually, the same holds for shrinkwrapped commercial software. Shrinkwrap does not mean virus free. It is most frequently used in conjunction with the licensing language, i.e., breaking the shrinkwrap means agreement with the terms of the license. Package encapsulation to reduce the risk of loss or contamination and increase the probability that tampering will leave evidence is an incidental benefit.

² The Virus Simulation Suite is written and maintained by Joe Hirst at the British Computer Virus Research Center, 12 Guildford St, Brighton, East Sussex, BN1 3LS, England. He can be contacted at 0273-26105.

- Cascade
- Den Zuk
- Fu Manchu
- Ping Pong
- Jerusalem

These simulations could help develop viral activity recognition in users.

B. HARDWARE MEASURES

Virus protection, as in many areas of computer security, may be possible, but, certainly not foolproof, without hardware assistance. Many incidence of viral infection could be reduced or eliminated by the introduction of a few hardware elements of varying cost:

- write protect tabs
- tamper-proof shrinkwrap
- CD ROM

1. Write Protect Tabs

These devices are inexpensive and easy to use. Most computer systems, and certainly those manufactured in the last five years, implement disk write protection in hardware vice software. Since this hardware inhibits disk writing if the tab is in place, viruses can not corrupt a write protected disk. Write protect tabs should be on all floppies put in a system.¹

¹ The hard drive or a dedicated flopp could then be used for output.

Original distribution diskettes should be write protected as soon as removed from the shrinkwrap and either installed on the hard drive or diskcopied to other floppies to make a working copy of the software.

2. Tamper-proof Shrinkwrap

Unfortunately for the buyer, many software stores have the capability to re-shrinkwrap software which has been purchased and returned or used for showroom demonstration. While this makes the product once again available for sale, it does nothing to protect the buyer from diskettes which were infected by the machine on which they had been previously used. To preclude this unsuspected hazard, tamper-proof shrinkwrap, perhaps bearing the vendor's corporate logo, could be used. Opening and resealing of software could be immediately determined by inspection.

3. CD ROM

Since systems are infected by using infected software, the most certain means of preventing infection is using only software which is certified virus free. We must then ensure it is never unintentionally modified. One method is the exclusive use of software on CD-ROM. This optically read storage medium, while much slower and more error prone than electrically read magnetic media, is forever inscribed with the digital data representing program and data files. All program output would be directed to regular magnetic media. The drawback is the prohibition of user software customization.

C. SOFTWARE MEASURES

The last software method for combatting viruses includes two types of measures:

- virus scanners which search for viruses in memory and on disk
- software which authenticates user software prior to use from a known true baseline

1. Virus Scanners

Virus scanners are programs which search computer memory and disk storage looking for signature byte strings characteristic of known viruses. As previously discussed, there are three major categories:

- programs which prevent infection
- programs which detect infection after it occurs
- programs which detect pre-existing infection

2. Authentication Methods

Software, both commercial or government produced, is shipped from vendors to either retailers or users. While undergoing development, we assume that it is protected. However, while sitting in storage, or on the user system itself, we consider it vulnerable. (Spafford, 1990)

Given this, we may develop a software authentication process consisting of the following:

- the vendor generates digitally signed software
- the user verifies the signed software
- the user installs and customizes the software on his system
- the user digitally signs his installed copy

- the user's system checks the executing software using hardware and/or software authentication methods (Davida, 1989, p 313)

Hardware is generally more tamper resistant than software. The same holds true for hardware based authenticators. The actual authentication method used, however, will be decided based on the level of security (degree of trust) required and the dollars available for procuring this means.

The inherent vulnerability of software based security systems is often compensated for by requiring complex access procedures, sophisticated self verification, or booting from special software diskettes. Unfortunately, this may be more than the average user will religiously adhere to. Conversely, complex hardware based authentication systems could be subverted by implanting malicious code into the IC chips during the design process.¹ This necessitates testing and validation of authentication equipment by external means. Validation testing could be eased by simple authentication hardware design since:

- chip complexity would be low
- chips could be procured from multiple sources making tampering impractical
- multiple chip design would require the malicious code to be spread across devices to perform its task
- chips older than computer viruses could be used (Davida, 1989, p 315)

¹ Acknowledging this, the Department of Defense has recently started work on its own semiconductor plant. This plant would produce "clean" IC chips for use in sensitive equipment.

The key problem then is the verification of the verifying means. A possible solution is the use of cryptography and digital signatures to 'sign' a software package. The vendor signs his release or update and the user authenticates the signature. Once validated, the software can be installed, customized as required, and 'sealed' by the user using his own digital signature.

A program may be checked when installed and each time it is executed.¹ The question of authentication granularity, like the hardware or software authentication issue, is one of degree. Determination of the level at which to sign an executable, i.e., program, process, or instruction, is a function of the degree of trust required, the speed of the authentication process, the size of the executable, and the method used. (Davida, 1989, p 317)

¹ A complete check may not be possible if segments are only loaded as needed. In timesharing environments a process is at risk whenever it doesn't control the processor. Events such as paging, segmentation, or process swapping open the code to tampering. Re-authentication would then be required prior to regaining control.

V. AUTHENTICATION METHODS

This section examines the theory behind several software authentication methods broadly categorized as "digital signature" generators. A digital signature is a property, private to a user or process, that is used for signing messages. A reliable digital signature must satisfy five requirements:

- the signature must be unique and be able to be generated only by the user
- it must be computationally infeasible for authentic signatures to be generated (forged) by unauthorized users
- any receiver of a digitally signed message (and any dispute arbitrator) must be able to authenticate the signatures authenticity even after a considerable period of time
- digital signatories must not be able to deny an authorized signature as a forgery
- digital signatures must be cheap and easy to generate (Seberry, 1989, p 155)

The digital signature, since it's non-forgable, establishes sender authenticity equivalent to a written signature.

This work advocates implementation of initial authentication means with the least delay and cost possible. Therefore, the following methods, ordered from the simple to the complex, were reviewed since they represented existing, well understood, and easily implemented technology. They are:

- checksums
- cyclic redundancy codes
- encryption

- message authentication codes
- hybrid systems

A. CHECKSUMS

The simplest and fastest form of digital signature is the checksum. Checksums treat a file as a series of binary numbers. By summing all the binary numbers, a checksum or total value can be found. Changes to files can be readily determined by recalculating the checksum and comparing it with the previous result.¹

A file changed by a virus will have an altered checksum unless the virus tries to compensate. Unfortunately, it is quite easy to determine a file's stored checksum and determine an additive byte string which 'corrects' the corrupted file's checksum. Therefore, checksums do not meet requirement 2 for reliable digital signatures and are not sufficiently robust to detect changes in the face of a clever attack.

B. CYCLIC REDUNDANCY CODES

Most commonly used for error detecting during data transmission, polynomial or cyclic redundancy codes (CRC) can be used to detect changes caused by viruses. CRC codes are generated using input bit strings as a representation of a polynomial with coefficients of 0 or 1. A n-bit message or message segment is regarded as the coefficient list for a polynomial of degree n-1 with n terms, ranging from x^{n-1} to x^0 .

¹ This method is used by many operating systems including DOS to determine if reads and writes from and to disk have been performed correctly.

Polynomial arithmetic is performed modulo 2 and is functionally equivalent to an exclusive or operation. Long division, using modulo 2 subtraction, can only be performed if the dividend has as many bits as the divisor.

The CRC is computed by first selecting a generator polynomial of degree r , $G(x)$, with at least the high and low order bits set to 1.¹ Next, r zero bits are appended to the low order end of the n -bit message or message segment so it now contains $n+r$ bits and corresponds to the polynomial $x^r M(x)$. The generator bitstring (the coefficients of $G(x)$) is then divided into this new bitstring (the coefficients of $x^r M(x)$) using modulo 2 division. The remainder is then subtracted from the $x^r M(x)$ bitstring using modulo 2 subtraction. This resulting bitstring, polynomial $T(x)$, is the new message or message segment which is then sent. Upon receipt, it is divided by $G(x)$. If a remainder exists, a transmission error, or virus corruption, has occurred.

Since this technique is more sophisticated than a checksum, it can detect more subtle changes. Checksums rely solely on addition which is insensitive to the order of the added numbers and can not detect byte swapping. CRCs, however, can detect bit and byte swapping due to the position dependent logic used.² While most alterations will be caught, the CRC method will not detect errors or corruptions corresponding to polynomials containing $G(x)$ as a factor. If $G(x)$ contains two or more terms

¹ Three polynomials have become international standards for $G(x)$: CRC-12, CRC-16, and CRC-CCITT.

² CRCs are frequently used in floppy disk controllers in order to determine whether information has been correctly retrieved from disk.

(order $r > 1$), all single bit errors will be discovered. By making $x+1$ a prime factor of $G(x)$, we can catch all errors consisting of an odd number of inverted bits. Lastly, a CRC code with r check bits will detect all burst errors of length $\leq r$. Larger bursts have a $(1/2)^{r-1}$ probability of being unnoticed. (Tannenbaum, 1989, pp 210-211)

The bottom line is that requirement 2 for reliable digital signatures is not met.

C. ENCRYPTION

Cryptography uses ciphers to transform standard plaintext messages into secret ciphertext ones. This process, called encryption, and its reverse, decryption, are controlled by one or more cryptographic devices called keys.

Ciphers are typically classified in one of two general types: transpositions or substitutions. The former rearrange data bits or message characters while the latter replace bits, characters, or character blocks with previously chosen substitutes. Most computer applications, such as the National Institute of Standards and Technology (NIST) Digital Encryption Standard (DES) use both techniques¹.

Cryptoanalysis is the science of cipher breaking. A cipher is breakable if it is possible to determine either the plaintext or the key from the ciphertext, or the key from plaintext-ciphertext pairs. There are three basic methods used for attacking ciphers:

¹ Often implemented in a combination of hardware and software.

- ciphertext only
- known plaintext
- chosen plaintext (Denning, 1982, pp 2-3)

A ciphertext only attack requires the determination of the key solely from intercepted ciphertext, knowledge of the method of encryption, purloined plaintext, knowledge of the ciphertext subject matter, and testing for high frequency use words. The known plaintext attack method uses knowledge of some plaintext-ciphertext pairs.¹ Ciphers should be accepted for use only if they can withstand a known plaintext attack using an arbitrary number of plaintext-ciphertext pairs. The last method, chosen plaintext attack, is the most favorable for the cryptanalyst. This attack assumes possession of the ciphertext corresponding to selected plaintext.² A cipher is unconditionally secure if, no matter how much ciphertext is intercepted, there is not enough information to determine the plaintext. (Denning, 1982, pp 2-3)

While all ciphers are breakable given unlimited resources and time, we need not develop them to withstand open-ended attack. A cipher which is computationally infeasible to break will suffice. Computationally secure ciphers can not be broken by systematic analysis with reasonable resources and time.

¹ Encrypted computer source programs is an example. The ciphertext must contain certain keywords such as begin, end, loop, if, while, read, write, etc. An educated guess as to their placement can be made and the attack begun. The same can be done with the executable version of the software if the equivalent machine instruction byte codes are known.

² Databases have been identified as being particularly susceptible to this type of attack if users can insert records into the database and then observe the changes in the stored ciphertext.

1. Cryptographic Systems

A cryptographic system has five components:

- a plaintext message space, M
- a ciphertext message space, C
- a key space, K
- a family of encrypting transformations, $E_k: M \Rightarrow C$
- a family of decrypting transformations, $D_k: C \Rightarrow M$ (Denning, 1982, p 7)

A given encrypting or decrypting transformation is defined by an algorithm common to the entire family but using an unique key. For a given key K , the decrypting transformation D_k is the inverse of the encrypting transformation E_k , such that $D_k(E_k(M))=M$. The transformations E_k and D_k are described by parameters called the encryption key and decryption key, respectively. (Denning, 1982, p 8)

Cryptosystems must satisfy three general conditions:

- the encrypting and decrypting transformations must be efficient for all keys
- the system must be easy to use
- the security of the system should depend only on the secrecy of the keys and not on the secrecy of the algorithms E or D (Denning, 1982, p 8)

The first requirement is essential for a computer based application since data encryption and decryption, usually performed at transmission time, must not be a bottleneck. The second requirement implies it must be easy for the cryptographer to find a key with an invertible transformation. The last requirement implies that it should not be possible to break a

cipher simply by knowing the method of encryption or the algorithm used.
(Denning, 1982, p 8)

2. Reasons for Cryptography

There are two basic reasons for using cryptographic systems:

- secrecy
- authentication

a. Secrecy

Secrecy requires that plaintext data be impossible to determine from intercepted ciphertext:

- it should be computationally infeasible to determine the decrypting transformation D_f from intercepted ciphertext C , even if the corresponding plaintext M is known
- it should be computationally infeasible to determine plaintext M from intercepted ciphertext C (Denning, 1982, p 9)

Secrecy, therefore, requires only that the transformation D_f (the decryption key) be protected. (Denning, 1982, p 9)

b. Authenticity

Authenticity requires that a false ciphertext C' can not be substituted for a ciphertext C without detection:

- it should be computationally infeasible to determine the encrypting transformation E_f given C , even if the corresponding plaintext message M is known
- it should be computationally infeasible to find C' such that $D_f(C')$ is valid plaintext in the set M (Denning, 1982, p 9)

Authenticity, therefore, requires only that the transformation E_k (the encryption key) be protected. (Denning, 1982, p 9)

3. Types of Cryptosystems

Cryptosystems can be placed into two broad classes:

- symmetric
- asymmetric

a. *Symmetric Cryptosystems*

These systems are also called one-key systems since the encrypting and decrypting keys are the same or easily derived from each other. If both E_k and D_k are protected, both secrecy and authenticity are ensured. Secrecy can not be separated from authenticity since making either E_k or D_k public exposes the other. Therefore, all the requirements for secrecy and authenticity must hold in one-key systems. (Denning, 1982, p 10)

b. *Asymmetric Cryptosystems*

These systems are also called two-key systems with E_k and D_k such that it is computationally infeasible to determine one key from knowledge of the other. In another sense, E_k and D_k may be thought of as one-way functions since they are easy to compute but computationally infeasible to invert.¹ This allows publication of one without endangering the other. E_k is protected for authenticity, while protecting D_k would

¹ This means its very difficult to determine a plaintext message from the ciphertext.

ensure secrecy. This duality makes these cryptosystems ideally suited for creating digital signatures. (Denning, 1982, p 11)

D. MESSAGE AUTHENTICATION CODES

Although these functions use cryptographic means to create a message authentication code (MAC), I will discuss them separately from cryptosystems. These ciphers are primarily used in applications where the need is not the decryption of transmitted data but the determination of a correspondence between a message M and ciphertext C . This correspondence is tested by encrypting M and comparing the result with C .¹ Several types of MAC systems exist. I will examine the following:

- public keys cryptosystems
- message digests

1. Public Key Cryptosystems

The public key system, an asymmetric cryptosystem, allows two users to hold both a public and private key and communicate with each other knowing only the others public key.

User A has a public encrypting transformation (key) E_A which may be widely known, and a private decrypting transformation (key) D_A which is known only to him. While the public key is derived from the private key by a one-way transformation, it is computationally infeasible to find D_A (or

¹ Computer logon passwords may be encrypted in this method. Since they can not be decrypted, they are secure. Yet, when a user enters his password at logon time, the transformation is applied and the encrypted logon password is compared with the stored encrypted true password. If they match, logon is achieved.

its equivalent) from it. It is possible to apply these transformations or keys to ensure three different outcomes:

- secrecy
- authenticity
- secrecy with authenticity (Denning, 1982, pp 11-12)

a. Providing Secrecy

If A wishes to send B a message, and knows B's public key E_b , he can transmit M to B in secrecy by sending the ciphertext $C = E_b(M)$. On receipt, B decrypts C using his private key D_b getting the plaintext message $M = D_b(C) = D_b(E_b(M))$. While this process provides secrecy, it does not provide authentication since any user with access to B's public key could send a message to A or replace one with his own. (Denning, 1982, p 12)

b. Providing Authentication

For authentication, M must be transformed using A's private key $C = D_a(M)$. On receipt, B uses A's public key $M = E_a(C) = E_a(D_a(M))$. Authenticity is provided since only A can apply the private key. Unfortunately, secrecy is not provided since anyone with access to the public key can obtain the plaintext. (Denning, 1982, p 12)

c. Secrecy with Authentication

To achieve both secrecy and authentication, A and B must apply both sets of transformations. A first applies his private key on M to assure authenticity $C' = D_a(M)$. He then encrypts this with B's public key to provide secrecy $C = E_b(C') = E_b(D_a(M))$. (Denning, 1982, p 13)

2. Message Digests

The public key system can generate a digital signature by using the sender's private key. This results in a signed message which is, however, twice the size of the original. To alleviate this, most techniques make use of data compression before forming the signature. Here, we take a N bit message and break it up into units of n bits (where $n \ll N$). The transformation is then applied to each of these units with the results combined to form a signature of n bits called a message digest. The key is ensuring that computing identical digests for two different messages is computationally infeasible. (Seberry, 1989, p 156)

Two message digest generator methodologies follow:

- hash functions
- the RSA Signature Scheme

a. Hash Functions

These are one-way functions which take variable size input strings and return a fixed size unique output string. The primary use of hash functions is to determine if there have been any changes made to a file.

Application of one-way function F on a plaintext message M yields a fixed size hash code $H = F(M)$. If M is an executable program file, $H_0 = F(M)$ is its hash code. If M is altered in any way, a new hash code, $H_n = F(M')$, will result. Thus, tampering can be detected by comparing the new hash with the previously computed, and presumably correct, one.

The property of one-way hash functions allowing their use for authentication is the computational infeasibility of finding a second message M' that will generate the same hash code H as the original message M . Because of this, a relatively small H , 128 to 256 bits in length, can be used to authenticate very large files. (Merkle, 1989A)

Formally, secure one-way hash functions have four properties:

- F can be applied to an M of any size
- F produces a fixed size H
- given F and M , it is easy to compute $H = F(M)$
- given F , it is computationally infeasible to find a different input M' , such that $M \neq M'$ and $F(M) = F(M')$ (Merkle, 1989B)

Since practical application requires a known input size, the hash code is normally developed in two steps. First, function F_f is defined. It is similar to F but accepts only fixed size inputs. F_f is used repeatedly to construct F via bitwise concatenation. All properties of F hold for F_f with the exception of the fixed input size. Deducing M given H is now equivalent to determining the key given the plaintext and the ciphertext. (Merkle, 1989B)

b. The RSA Signature Scheme

The RSA scheme¹ uses both the public key cryptosystem and data compression to form a digital signature. Assuming A wishes to send B a signed message M , he first shortens it using compression to arrive at the message digest $M_f = F(M)$. Next, A encrypts the digest using his

¹ Named as a supported algorithm in the NIST/OSI Implementor's Workshop Agreements of Dec 1989.

private key to obtain the signature $M_s = D_A(M_f) = D_A(F(M))$. The message and signature pair (M and M_s) are then forwarded to B .

On receipt of M and M_s , B reproduces the message digest by two different methods. First, the digest is recreated from the signature by applying A 's public key $M_f = E_A(M_s)$. Then B produces his own copy of the digest using the compression function (since it's public). If the digest obtained from A 's signature equals the digest B created, the signed M is accepted as authentic.

E. HYBRID SYSTEMS

These systems generally use a combination of methodologies to optimize the speed of computer processing versus level of trust. For example, a DES based system makes major demands on a personal computer due to the complex mathematics required for encryption.¹ In order to speed the processing from the user's viewpoint, a hybrid mixture of DES and CRC, or other means, may be used. This usually results in a small percentage of a file being examined with a sophisticated MAC, and the remainder examined with a high speed CRC algorithm.² Sophisticated cryptographic techniques are used to assure that attackers can not predict which bytes are examined by each method. Additionally, results of all cryptographic calculations are carried forward into all subsequent calculations. This

¹ In fact, actual bit manipulation is most often implemented in hardware for both speed and security concerns.

² Perhaps the portion examined by the MAC would be the code sections most frequently modified by viruses.

results in a digital signature that is faster than a DES MAC and stronger than a CRC. (Bosen, 1989, pp 7-9)

VI. PRACTICAL SOFTWARE AUTHENTICATION

This section examines several commercially available products which make practical use of the theory previously discussed.

- RSA's MD4 algorithm
- RSA's CHECK and SIGN programs
- Enigma Logic's VIRUS-SAFE

A. MD4

This package, programmed and maintained by Ronald Rivest of RSA Data Security, has been placed in the public domain for review and possible adoption as a standard.¹ MD4 inputs a message of arbitrary length and produces a 128 or 256 bit message digest.

It is considered computationally infeasible to produce two messages having the same message digest, or any message having a specified target message digest.² Although MD4 is relatively new, the security provided should be sufficient for implementing very high security hybrid digital signature schemes based on MD4 and a public-key cryptosystem. (Rivest, 1990)

¹ As have been the two previous MD family algorithms MD2 and MD3. These were circulated throughout the industry as INTERNET RFC 1113 and 1114. MD4 has been distributed as RFC 1115.

² RSA conjectures the difficulty of deriving two messages having the same message digest is on the order of 2^{64} operations, and that of any message having a given message digest is on the order of 2^{128} operations.

The C version of the MD4 algorithm, listed in Appendix G, is coded for a 32-bit word/8-bit byte machine and runs at 1450, 70, and 32 kBytes per second on a SUN Sparc station, DEC MicroVax II, and 20MHz 80286, respectively. (Rivest, 1990)

B. SIGN AND CHECK

RSA Sign & Check are two programs which allow users to sign files with non-forgable digital signatures and to check a given signature's validity. These signatures are based on the highly trusted RSA public key cryptosystem which has withstood intensive mission critical commercial use as well as fourteen years of vigorous challenge from the academic and scientific communities.

C. VIRUS-SAFE

This product relies on three different methods for producing MACs:

- cyclic redundancy codes
- American National Standards Institute (ANSI) standard X9.9¹
- International Standards Organization (ISO) standard 8731-2²

VIRUS SAFE allows several methods of operation to optimize security versus performance.³ The frequency and thoroughness of file examination

¹ ANSI X9.9 describes a way of using DES to calculate a MAC which is believed impossible to forge.

² ISO 8731-2 is currently used in the international banking community to authenticate funds transfer.

³ The thoroughness of file examination which makes unauthorized modification impossible to remain undetected takes time and decreases productivity.

can be optionally set either during software installation or execution such as:

- whether or not a file should be examined at all
- when to examine the file (at boot or execution time)
- how frequently to examine the file (every time, every other, etc)
- how thoroughly to examine the file (the ratio of MAC to CRC)

For example, someone using a word processor could opt for thorough examination every 15th execution instead of each time it's used. A programmer who infrequently uses a debugger may want to have it examined every time it is used. The infrequent use means the productivity impact is small, even if a thorough examination is performed. Other non sensitive program and data files can be examined in a more routine manner, such as, using a high speed algorithm when the computer is booted. This approach could be up to ten times as fast, but, not offer the level of security of a MAC. (Bosen, 1989)

On a 10 MHz AT, 100 kBytes can be authenticated in 3.2 seconds using a hybrid DES and CRC algorithm. (Bosen, 1989)

VII. CONCLUSIONS AND RECOMMENDATIONS

Earlier, I asked, "How do we provide protection from viral attack?". I concluded the appropriate question should be, "How do we prevent the loading of infected software?". In retrospect, the first question allows us to define the foundation for the paradigm to answer the second.

Serious protection from viral attack should be implemented as a three-fold program:

- user training
- virus detection and removal
- software authentication

The program, however, is best executed in two phases:

- confidence building
- assurance building

A. CONFIDENCE BUILDING

These measures instill confidence in the user and organization that computer viruses are not omnipotent programs written by omniscient programmers. Instead, they are damaging, but common, nuisances which can be recognized and defeated with a little effort and understanding. Confidence building measures would entail the first two-thirds of the three step program:

- user training
- virus detection and removal

1. User Training

A solid foundation for further anti-viral efforts must be laid by training computer users in two areas:

- basic precautions
- viral recognition

The basic precautions will provide good user practices to reduce the impact of viral damage, slow infection spread, and, hopefully, eliminate new infections. A user's ability to recognize viral symptoms, either by suspicious activity or visual or audible cues is important. Recognition, before the initiation of the damage sequence, will be damage and infection limiting. Use of virus simulations will provide safe and realistic recognition patterns.

2. Virus Detection and Removal

Once the user has developed sound prevention and identification skills, active use of anti-viral products will help ensure a virus free system. These products should be implemented in the following order:

- identification tools
- detection or prevention tools

The identification product will ensure all existing infections are identified and removed. Then the virus detection or prevention product can be installed to preclude new infections.

These measures should be used as follows:

- daily random checks of on-line software as a first line of defense against viral infection
- periodic deliberate checks against an off-line protected baseline to verify the random verification means

B. ASSURANCE BUILDING

While the above measures will provide a strong damage and infection prevention program, additional means are available which could strengthen prevention measures and cover "apses"¹ in their use. These assurance building measures should make use of the inherent security of digitally signed software. This means, when preparing software procurement contract specifications, organizations should require that vendors certify their software is virus free and that it has been digitally signed using a "approved" product^{2,3}.

1. Software Authentication

Several acknowledged methods currently exist for simple and inexpensive software authentication:

¹ Realistically, most people will not write protect floppy disks or use virus checkers religiously.

² This could be done in conjunction with the National Institute of Standards and Technology which is currently developing a standard digital signature for non-repudiation. Likewise, vendors such as RSA provide respected products and act as agent for establishing and maintaining the unique digital signature applied by each software package sold.

³ The digital signature should be produced on the vendor's development system since it is assumed to be uninfected. This is not unrealistic since infections of commercial programs to date have been traced to the duplication hardware and not the development system.

- use of a public key type signature product such as Sign & Check
- use of a one-way hash functions such as MD4
- use of multi-function products such as VIRUS-SAFE

The relative advantages and disadvantages of these methods include:

- One-way hash functions do not require any information to be kept secret. Public key systems require one key be kept secret.¹
- The public key system would not require creation of substantiating documentation. The digital signature is either valid or not. If its not, the software is not loaded. One-way hash functions, however, require that a substantial paper trail of software² valid hash codes be maintained. Conformity of the hash code generated upon software receipt, with the approved hash code maintained in the valid hash codes documentation, is then required prior to installation.
- The public key system is much simpler in application than the one-way hash function method.
- Public key system licensing costs several hundred dollars per site. The one-way hash function code is in the public domain.
- A multi-feature program could be used in the interim to validate software while waiting for vendors to release digitally signed software.

¹ In the form of the digital signature creation algorithm. This is kept secret by the vendor.

² This includes all versions and mini-releases.

³ For this work, I assume that NAVELEXSECCEN, the Navy's anti-virus command, will create and maintain this paper trail. They will select the Navy standard hash function, apply it to all mission support software in Navy's inventory, and release a NAVELEXSECCEN NOTICE detailing the software package, version, manufacturer, and hash code. As new software is procured or updates received, NAVELEXSECCEN will update and reissue their NOTICE. Of course, hashing for new software could be required by the procurement contact to relieve NAVELEXSECCEN of all but the initial effort.

C. THE BOTTOM LINE

A reliable anti-virus product must be robust enough to ensure its own integrity, and sophisticated enough to check all executable files without exception.

While evaluating schemes for detecting and preventing viral spread, it is important to remember that viruses use the same system capabilities available to users. Many products and precautions may be used to slow or stop infection spread. Each, however, tends to reduce the computer's utility. As long as we desire flexibility, viruses will be able to exploit legitimate system capabilities. As viral programmers become more experienced and develop new techniques, distinguishing between legitimate and viral activity will become an increasingly difficult problem.

This requires more stringent protection and verification schemes. Their strength, however, should lie in the verification process and not in protection or secrecy of the method. Since most, if not all, of the anti-viral products available today have loop holes¹, we must shift our reliance from these products to a computationally secure methodology for absolute virus free environments.

Since viral infection can be traced to the use of infected software², the only efficient way to preclude infection is to prohibit the use of tainted programs. The only trustworthy means of prohibition is to require

¹ Whether it be software interrupt monitoring in the prevention product, assumed clean baselines for the detection product, or known signature byte strings for the identification product, viruses can exploit the weaknesses of even the most sophisticated anti-viral program.

² While the current generation of microcomputer viruses live mostly in executable images, this may not necessarily be true in the future.

that vendors cooperate with users in a program of software authentication using digital signatures.

VIII. APPENDICES

A. EXAMPLES OF DOS VIRAL INFECTION IN THE US

INFECTED SOFTWARE	REPORTING LOCATION	DATE	VIRUS
Unlock Masterkey	Kennedy Space Center	Oct 89	Vienna
SARGON III	Iceland	Sep 89	Cascade (1704)
ASYST RTDEMO02.EXE	Fort Belvoir	Aug 89	Jerusalem-B
Desktop Fractal Design System	Various	Jan 90	Jerusalem (1813)
Census Bureau 1988 Election: City & County Data Bank	Government Printing Office/US Census Bureau	Jan 90	Jerusalem-B
Northern Computers	Iceland	Mar 90	Disk Killer

B. CHRONOLOGY OF A VIRUS AS TOLD BY IT'S AUTHOR

The author of the Dark Avenger virus has distributed it's source as well as a program, DOCTOR, to remove it. DOCTOR contains the following:

DOCTOR QUICK! Virus Doctor for the Eddie Virus Ver 2.01 10-31-89
(c) 1988-89 Dark Avenger. All rights reserved. DOCTOR /? for help

It may be of interest to you to know that Eddie (aka "Dark Avenger") is the most widespread virus in Bulgaria for the time being. However, I have information that Eddie is well-known in USA, W. Germany and USSR too.

I started writing the virus in early September 1988. In those times there were no any viruses written in Bulgaria, so I decided to write the first Bulgarian virus. There were some different Eddie's versions:

VERSION 1.0, 31-OCT-1988

This version established the most important features of the Eddie virus. Staying resident into high end of memory, it was infecting .COM and .EXE files, but only when executing them. INT 13 hadn't been handled in any way. This version was damaging infected files only, rather than infected disks. Also, there weren't any messages in it (I still wasn't choosed a name for it).

VERSION 1.1, 16-DEC-1988

In December I've decided to enhance the virus. This version could infect files during their opening. For that reason, a read buffer was allocated in high end of memory, rather than using DOS function 4Bh when needed. The disk was destroyed instead of the infected files.

VERSION 1.2, 19-DEC-1988

This added a new feature that causes (for example) compiled programs to be infected at once if the virus is resident. Also, the "Eddie lives..." message was added (can you guess why exactly "Eddie"?).

VERSION 1.31, 3-JAN-1989

This became the most common version of Eddie. A code was added to find the INT 13 rom-vector on many popular XT's and AT's. Also, other messages were added so its length would be exactly 1800 bytes. There was a subsequent, 1.32 version (19-JAN-1989), which added self-checksum and other interesting features that was abandoned because it was extremely buggy. In early March 1989 version 1.31 was called into existence and started to live its own life to all engineers' and other suckers' terror. And, the last

VERSION 1.4, 17-OCT-1989

This was a bugfix for version 1.31, and added some interesting new features. Support has been added for DOS 2.x and DOS 4.x. For further information about this (the most terrible) version, and to learn how to find out a program author by its code, or why virus-writers are still not dead, contact Mr. Vesselin Bontchev (All Rights Reserved).

So, never say die! Eddie lives on and on and on... Up the irons!

C. KNOWN VIRUS INFECTION AND DAMAGE CHARACTERISTICS

The information below, provided by John McAfee at CVIA, details the major characteristics and number of variants, in parenthesis, of the known IBM PC compatible virus strains.

A Infects Fixed Disk Partition Table--+											Directly or Indirectly:										
9 Infects Fixed Disk Boot Sector-----+											+-----Affects Performance										
8 Infects Floppy Diskette Boot-----+											+Corrupts Programs/Overlays										
7 Infects Overlay Files-----+											+---Corrupts File Linkages										
6 Infects EXE Files-----+											+---Corrupts Data Files										
5 Infects COM files-----+											+Corrupts Boot Sector										
4 Infects COMMAND.COM-----+											+-----Formats Disk										
3 Install Self in Memory+																					
2 Self-Encryption-----+																					
1 STEALTH Techniques+																					
	1	2	3	4	5	6	7	8	9	A	1	2	3	4	5	6	Increase in Infected Program Size				
Virus	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V				
Azusa	.	.	x	.	.	.	.	x	.	x	x	.	x	x	x	.	N/A				
Lazy	.	.	x	x	x	.	.	.	.	.	x	x	.	.	.	.	720				
V-555	.	.	x	x	x	x	x	.	.	.	x	x	.	.	.	.	555				
Phantom	.	.	x	x	x	.	.	.	.	.	x	x	.	.	.	.	2253				
V-299	.	.	.	.	x	.	.	.	.	.	x	x	.	x	.	.	299				
Cancer	.	.	.	.	x	.	.	.	.	.	x	x	.	x	.	.	1480				
1575/1591	.	.	x	x	x	x	.	.	.	.	x	x	x	.	.	.	vary				
USSR 492	.	.	.	.	x	.	.	.	.	.	x	x	.	.	.	.	492				
USSR 1049	.	.	x	.	x	x	.	.	.	.	x	x	.	.	.	.	1049				
Skism	.	.	x	.	x	x	x	.	.	.	x	x	.	.	.	.	1815				
Holocaust	x	.	x	x	x	.	.	.	.	.	x	x	x	x	.	.	3784				
Stone-90	.	.	.	x	x	.	.	.	.	.	x	x	.	.	.	.	961				
903	.	.	x	x	x	.	.	.	.	.	x	x	.	.	.	.	903				
Dir-Vir	x	.	x	x	x	.	.	.	.	.	x	x	.	x	.	.	691				
Hybrid	.	.	.	x	x	.	.	.	.	.	x	x	x	.	.	.	1306				
IKV528	.	.	.	x	x	.	.	.	.	.	x	x	.	.	.	.	528				
Iraqi Warrior	.	.	.	x	x	.	.	.	.	.	x	x	x	x	.	.	777				
Little Pieces	.	.	x	.	x	x	.	.	.	.	x	x	.	.	.	.	1374				
Saddam	.	.	x	x	x	.	.	.	.	.	x	x	x	x	.	.	919				
Monxla-B	.	.	.	x	x	.	.	.	.	.	x	x	x	.	.	.	535				
Plague	.	.	.	.	x	x	.	.	.	.	.	.	.	.	.	.	Overwrites				
Polish 217	.	.	.	x	x	.	.	.	.	.	x	x	.	.	.	.	217				
Sentinel	.	.	x	x	x	x	.	.	.	.	x	x	x	x	.	.	4625				
Swiss 143	.	.	.	x	x	.	.	.	.	.	x	x	.	x	.	.	143				
USSR-311	.	.	.	.	x	.	.	.	.	.	x	x	.	.	.	.	321				
Voronezh	.	x	x	x	x	x	.	.	.	.	x	x	.	x	.	.	1600				
V-961	.	.	x	x	x	.	.	.	.	.	x	x	.	.	.	.	961				
USSR-830	.	.	x	x	x	.	.	.	.	.	x	x	.	.	.	.	830				
USSR-529	.	.	x	x	x	.	.	.	.	.	x	x	.	.	.	.	529				
USSR-516	.	.	x	x	x	.	.	.	.	.	x	x	.	.	.	.	516				
USSR-2144	.	x	x	x	x	x	.	.	.	.	x	x	x	x	.	.	2144				
USSR-1049	.	.	x	x	x	.	.	.	.	.	x	x	.	.	.	.	1049				
USSR (3)	.	x	.	.	.	x	.	.	.	.	x	x	.	.	.	.	575				
Tiny-133	.	.	.	x	x	.	.	.	.	.	x	x	.	.	.	.	133				
Sverdlov	.	.	x	x	x	x	.	.	.	.	x	x	.	.	.	.	1962				
Label	.	.	x	x	x	.	.	.	.	.	.	.	.	.	.	.	Overwrites				
Kukaturbo	.	.	x	x	x	.	.	.	.	.	.	.	.	.	.	.	Overwrites				

Justice	. . x x x	x x	1242
Hymn	. . x x x x x	x x . x	642
Happy New Year	. . x x x x x	x x	1865
Destructor	. . x x x x x	x x	1150
Leapfrog Virus	. . x ? x	x x . x	516
MGTU Virus (2)	. . . x x	x x . x	273
Nina	. . x x x	x x . x	256
Lozinsky	. . . x x	x x . x	1023
BeBe	. . . x x	x x . x	1004
Best Wish	. . . x x x x	x x . x	1024
Beeper (2)	. . x . x	x x . x	492
Parity	. . . x x	x x . x	441
Trust Me	. . x x x	x x . x	417
USSR-948	. x . . x x x	x x . x	948
USSR-711	. x . . x	. x . x	711
USSR-707	. x . . x	. x . x	707
USSR-696	. x . . x	. x . x	696
USSR-600	. x . . x	. x . x	600
USSR-394	. x . . x	. x . x	394
USSR-257	. x . . x	. x . x	257
USSR-256	. x . . x	. x . x	256
Christmas Violator	. . . ? x	x x . x	????
Off Stealth	x . x x x x x	x x . x	1689
Jeff	. . . x x	x x . x . x . .	928
Bloody!	. x x x . x	x . . . x . . .	n/a
ZeroHunt	x x x . x	x x . x	n/a
Music Bug (3)	. . x x x .	x . . . x . . .	n/a
Dot Killer	. . x x x	x x	944
Father Christmas	. . . x x	x x	1881
3445	x x x . x x	x x x x	3445
Mirror (2)	. . x . . x	x x	928
Polish-2	. . x x x	x x . x	512
Polish 217	. . . x x	x x . x	217
Happy Day	. . . x x	x x	453
Monxia	. . . x x	x x	939
USSR	. x . . . x	x x	575
Polimer	. . . x x	x x . x	512
DataLock	. . x . . x	x x	920
Carioca (2)	. . x . . x	x x	951
529	. . x x x	x x . x	529
Spyer	. . x . . x x x	x x	1181
Taiwan4	. . x x x x x	x x . x	2576
Keypress (3)	. . x x x x	x x . x	1232
Casper	. x . . x	x x x x	1200
1605	. . x x x x	x x x x	1605
Violator (6)	. . . x x	x x . x	1055
Blood-2	 x	x x . x	427
Wisconsin	. x . . x	x x . x	825
Christmas-J	. . x x x x	x x	600
Austria (3)	. . . x x x		Overwrites
Leprosy-B	. . . x x x		Overwrites
Whale (3)	x x x x x x x	x x x x	9216
Invader (4)	. x x . x x x x x . .	x x x x x . . .	4096
Scott's Valley	. x x . x x x	x x x x	2133
Anarkia (2)	. . x . . x x x	x x . x	1813
Black Monday (2)	. . x x x x x	x x x x	1055
Nomenclature (4)	. . x x x x x	x x . x	1024
Anthrax - Boot (2)	. . x x . .	x x . x	n/a
Anthrax - File (4)	. . x x x x	x x . x	1206
651	. . x	x x . x	651
Paris	. . . x x x x	x x x x	4909
Leprosy (5)	. . x x x x x		Overwrites
Mardi Bros. (3)	. . x x x . .	x . . . x . . .	n/a
1253 - Boot	. . x x x x	x x x x	n/a
1253 - COM	. . x x x	x x x x	1253
AirCop (3)	. . x x . . .	x . . . x . . .	n/a
400 (5)	. . x . x	x x . x	vary
P1 (6)	. x x . x	x x x x	vary

Ontario	. x x x x x	x x . x . .	varv
1226 (3)	. x x x x x x . . .	x x . x . .	1226
V2100 (2)	. . x . x x	x x x x . .	2100
Plastique (9)	. . x x x x x	x x . x . .	3012
Wo man (2)	. . x x x x	x x	2064
Doom2	. . x . x x	x x x x . .	2504
Flip (4)	. x x x x x x . . .	x x x x . .	2343
Fellowship (3)	. . x . . x	x x x x . .	1022
Flash	. . x x x x	x x x x . .	688
1008	. x x x x	x x x x . .	1008
Stoned-II	. . x x . x	x . x . x .	n/a
Taiwan3	. . x x x x x	x x x x . .	2905
Armagedon (3)	. . x x x	x x	1079
1381	 x x	x x	1381
Tiny (13)	. . . x x	x x	163
Subliminal (3)	. . x x x	x x	1496
Sorry	. . x x x	x x	731
RedX (2)	. . . x x	x x	796
1024 (2)	. . x x x	x x	1024
Joshi (4)	x . x x x x	x . . x x .	n/a
Microbes	. . x x x .	x . . x x .	n/a
Print Screen (2)	. . x x x .	x . . x x .	n/a
Form (2)	. . x x x .	x . . x x .	n/a
July 13th	. x . . . x	x x x x . .	1201
5120 (3)	. . . x x x x	x x x x . .	5120
Victor (2)	. . x x x x x	. x x x . .	2458
JoJo (3)	. . x . x	x x	1701
W-13 (4)	 x	x x	532
Slow (5)	. x x . x x x	x x x . . .	1721
Frere Jacques	. . x . x x x	x x	1811
Liberty (2)	. . x x x x x	x x	2862
Fish-6 (2)	x x x x x x x	x x x . . .	3584
Shake	. . x x x x x	x x	476
Murphy	. . x x x x x	x x	1277
V800 (3)	x x x . x	x x x . . .	none
Kennedy (3)	. . x . x	x x	308
8 Tunes/1971 (2)	. . x . x x x	x x	1971
Yankee-2	 x x	x x	1961
June 16th	. . . x x	x x x . . x	1726
XAl	. x . . . x	x x x . . x	1539
1392	. . x x x x	x x x . . .	1392
1210	. . x . x	x x x . . .	1210
1720 (3)	. . x . x x x	x x x . . x	1720
Saturday 14th (3)	. . x . x x x	x x x . . x	685
Korea (4)	 x x	x . . . x .	n/a
V . . . (5)	 x	x x x . . .	.074
I Vir	 x	x x x . x .	3880
S . . . (3)	. . x . x	x x x . . .	2000
V . . . (3)	. . x x x x x	x x x . . .	2000
1559	. . x x x x	x x x . . .	1554
512 (5)	x . x x x	x x x . . .	none
EDV (2)	x . x x x x	x . . . x .	n/a
Joker	. . x x x	x x	
Icelandic-3	. . x . . x	x x	853
Virus-101	. x x x x x x	. x	2560
1260 (3)	. x . . x	. x	1260
Perfume (2)	 x	. x	765
Taiwan (3)	 x	. x	708
Chaos	. . x x x .	x . . x x x	n/a
Virus-90	. . x . x	. x	857
Oropax (5)	. . x . x	x x	2773
4096 (4)	x . x x x x x	x x x x . .	4096
Devil's Dance (2)	. . x . x	x x x x . .	941
Amstrad (5)	 x	. x	947
Payday (2)	. . x . x x x	. x	1808
Datacrime II-P	. x . x x x	. x . . . x	1917
Sylvia/Holland	 x	. x	1332
Do-Nothing	. . x . x	. x	608

Sunday (4)	. . X . X X X . . .	X X	1636
Lisbon (2)	 X	. X	648
Typo/Fumble	. . X . X	X X	867
Dbase	. . X . X	X X . X . .	1864
Ghost Boot	. . X X X .	X . . . X .	n/a
Ghost COM	 X	. X . . X .	2351
New Jerusalem	. . X . X X X . . .	X X	1908
Alabama (3)	. . X . . X	X X X . . .	1560
Yank Doodle (6)	. . X . . X X	X X	2885
2930	. . X . . X X	. X	2930
Ashar	. . X X . .	 X .	n/a
AIDS (4)	 X		Overwrites
Disk Killer (4)	. . X X X .	X X . X X X	n/a
1536/Zero Bug	. . X . X	X X	1536
MIX1	. . X . . X	X X	1619
Dark Avenger (4)	. . X X X X X . . .	X X X . . .	1900
3551/Svslock	. . X . . X X	. X . X . .	3551
VACSINA (5)	. . X . . X X X . . .	X X	1206
Ohio	. . X X . .	 X .	n/a
Typo Boot	. . X X X .	X . . . X .	n/a
Swap Boot	. . X X . .	 X .	n/a
Datacrime-2 (2)	. . X . . X X	. X . . . X	1514
Icelandic II	. . X . . X	X X	661
Pentagon	 X	 X .	n/a
Traceback (3)	. . X . . X X	. X	3066
Datacrime-B	. . X . . X	. X . . . X	1168
Icelandic (2)	. . X . . X	X X	642
Saratoga	. . X . . X	X X	632
405	 X		Overwrites
1704 Format	. . X X . X	X X . . . X	1704
Fu Manchu (4)	. . X . . X X X . . .	X X	2086
Datacrime (2)	. . X . . X	. X . . . X	1290
1701/Cascade	. . X X . X	X X	1701
CASCADE-B (9)	. . X X . X	X X	1704
Stoned (5)	. . X X . X	X . X . X .	n/a
1704/CASCADE	. . X X . X	X X	1704
Ping Pong-B (2)	. . X X X .	X . . . X .	n/a
Den Zuk (3)	. . X X . .	X . . . X .	n/a
Ping Pong (5)	. . X X . .	X . . . X .	n/a
Vienna-B	 X	. X	648
Lehigh	. . X X	. X . . . X	Overwrites
Vienna/648 (23)	 X	. X	648
Jerusalem-B	. . X . . X X X . . .	X X	1808
Alameda (2)	. . X X . .	 X .	n/a
Friday 13th COM	 X	. X	512
Jerusalem (17)	. . X . . X X X . . .	X X	1808
SURIV03	. . X . . X X X . . .	X X	
SURIV02	. . X . . X	X X	1488
SURIV01	. . X . . X	X X	897
Brain (3)	. . X X . .	 X .	n/a

Total Known Viruses - 481

LEGEND:

Size Increase:

N/A - Virus does not attach to files.
None - Virus does not change size (attaches to file end)
Overwrites - Virus overwrites beginning of file, no size change
All Others - Length in bytes a file will increase when infected.

Characteristics:

x - Yes
. - No

D. ANTI-VIRAL PRODUCT MINIMUM CAPABILITIES LIST

The minimum capabilities in the sections below are recommended by John McAfee at CVIA. They can be used to provide a rough cut on prospective anti-viral products. Acceptable products should then be evaluated using the testing criteria in Appendix E.

1. Infection Prevention Products

- Differentiate between activities initiated by the user and activities carried out autonomously by programs. Users may delete or update program files or operating system segments. An application program, on the other hand, should not, under normal circumstances, modify another application program, an operating system program, or the system's boot sector. This is indicative of viral activity;
- Provide few false positives, or false alarms. Users become habituated to frequent false alarms and tend to overlook a valid virus warning when it does occur;
- Run with other memory resident programs. Infection prevention programs are all memory resident and they modify a large number of software interrupts. This gives such programs a propensity for crashing or hanging the system when running concurrently with other memory resident programs;
- Protect against modifications to all executable data, including: the system's boot sector, device drivers, operating system modules, including hidden file programs, and all application programs;
- Provide an easily accessible enable/disable switch. Many instances will occur where the checking process will need to be temporarily suspended;
- Provide the ability to selectively protect or ignore specific programs or specific areas of the system. This will reduce the number of false alarms when running programs which violate the "rules" imposed by the product;
- Provide the ability to freeze virus activity when it is detected, and prevent the illegal access from continuing. This is mandatory to prevent the virus from infecting the system;
- Run without noticeably degrading system performance. Memory resident programs have a tendency to increase system overhead and thus slow down the system. A well designed product should cause no more than a 5% degradation in system performance;
- Monitor and protect all attached read/write devices. All attached devices that can be written to are potential virus targets. The prevention product should protect all such devices; and,
- Selectively prevent interrupt level I/O and non-standard calls for I/O service (interrupt level requests). Since doing so increases the false alarm rate, the user should have the choice of allowing or disallowing such calls.

2. Infection Detection Products

- Detect characteristic viral modifications to executable data, including: the system's boot sector, system device drivers, operating system modules, including hidden file programs, and all application programs;

- Allow the user to selectively exclude specific programs or areas of storage from checking. This will allow programs or directories that undergo frequent change to avoid causing error messages;
- Perform global check functions in a timely fashion. If the check function is executed at boot time, for example, it should add no more than 10 seconds to the boot sequence for each 50 programs on the disk that must be checked;
- Provide automatic checking. The check function should execute at least each time the system is powered on or re-booted. Some systems provide a clock function so that the system can be checked automatically at user specified time intervals;
- Stop the system, provide a visual and audible warning, and wait for user directions if a potential virus is detected; and,
- Display the names of all programs or system areas that have become infected.

3. Infection Identification Products

- Identify and remove multiple virus strains;
- Provide information to allow the user to determine the diagnosis accuracy. Modified viruses can sometimes only be detected by cross referencing many different characteristics. The product should provide the degree of certainty, or other information that can be used to determine a course of action, for any questionable virus;
- Identify and report infected system areas and the extent of infection;
- Inform the user of the anticipated degree of success for removal. Depending on the length of time since infection, removal may or may not be possible. The product should inform the user of possible options including automatic removal or erasure of the affected system element;
- Scan and remove infection from all attached devices including floppies, fixed and removable hard disks, and tape devices;
- Automatically scan all subdirectories;
- Flag all system areas where removal was incomplete. These areas must be manually dealt with after the program finishes; and,
- Prevent self infection during the identification and removal process. An infected identification product will run the risk of infecting every system on which it is used.

E. ANTI-VIRAL PRODUCT EVALUATION PROCEDURES

The test procedures in the sections below are recommended by John McAfee at CVIA. While not scientifically rigid, they will provide additional performance information not otherwise obtainable. Any product that performs well in testing will provide some degree of real protection.

1. Infection Prevention Products

- Install the antiviral product;
- Test the product's ability to protect general executable programs from being modified. Create a temporary subdirectory and copy your word processor into it. Create two output text files named TEST, one with a .EXE extension and the other with a .COM extension. Then attempt to update the file using the word processor. The antiviral program should flag both the creation and the update as a potential infection. Repeat these steps for the system files (IBMBIO.COM, IBMDOS.COM, and COMMAND.COM) as well as all device drivers. Repeat each of these steps using a floppy diskette as the output device, instead of the hard disk subdirectory. The same results should occur.
- Test the product's ability to prevent interrupt level I/O. First copy the FORMAT routine to a file named TEST.COM. Run TEST and format a floppy diskette in the A or B drive. The antiviral program should prevent the format and flag the attempt.
- Test the use of operating system commands. User commands are frequently, and erroneously, flagged by antiviral products when they instigate operations that mimic virus activities. Using COPY, DELETE and RENAME commands, copy an executable program into a different directory, rename it to another EXE or COM file name, and then delete it. None of the three operations should be flagged by the antiviral program.
- Verify that the above functions would be stopped if performed by a program, rather than by the user. Using any application utility program that has copy, rename and delete functions, repeat the above series of steps. The antiviral program should prevent and flag all three attempts as potential viral activities.
- Test self modification. Many programs modify their own executable modules at some point. The antiviral program should not flag or prevent this. To test this, copy your word processor executable module to a backup file. Then run the word processor, create a dummy document, and then save it to the name of the executable word processor module. The antiviral program should allow the modification. After this test, copy the saved version of the program back to its original name.
- Test for detection of boot sector modification. Using any utility that allows reading and writing the boot sector, read the boot sector and write down the contents of the first byte. Change the first byte to 00 and attempt to write the sector back to disk. The product should prevent the attempt. If the product fails, replace the original contents of the first byte and re-write the boot sector. The re-write should be performed prior to shutting down or re-booting the system.
- Test for memory residence. Many viruses modify the original structure of programs so that they remain memory resident after they terminate. The antiviral product should detect any attempt to remain resident. To test this feature, merely take any normally memory resident program, such as SIDEKICK or CACHE, and rename it to the file TEST.COM (or .EXE, depending on the program). Run TEST. The product should catch the program and display a warning message.

2. Infection Detection Products

- Test for detection of boot sector replacement. Using a disk utility, create a safe unique boot sector by blanking out the "Boot Failure" message. Then install the detection product you wish to test. Next, replace the entire boot sector using the SYS command. Then execute the check function of the product you are testing. The product should warn that the new boot sector is a replacement.
- Test for detection of boot modification. Next, re-install the detection product. Then modify the boot sector randomly using the disk utility. Run the check routine. The product should warn that the boot sector has been modified. (When finished with this step, perform the SYS command again, or use the disk utility to return the boot sector to its original state).
- Test for detection of program deletion. Copy a number of COM and EXE files to a temporary directory and then delete the originals. Run the detection check function. The product should identify each of the missing programs.
- Test for detection of program modification. Copy the programs back from the temporary directory to their original directories. Using your disk utility, modify the first byte of each of the COM programs. Modify the entire first 500 bytes of the EXE programs. Run the check program. Each modification should be detected. At this point you should replace each of the modified programs from the original programs stored in the temporary directory.
- Test for detection of program replacement. Replace an application program with the original from the distribution diskette. Then modify the program as above. The check function should still catch the modification.
- Test for detection of system modification. Using a disk utility, copy IBMBIO.COM, IBMDOS.COM, and COMMAND.COM to backup files. Randomly modify each of the original files, using the disk utility, by changing only one byte in each. Run the check routine to determine that the modifications have been detected. Perform this step multiple times with different modifications.

3. Infection Identification Products

- The first steps are to isolate the infected system from all others, and to acquire clean, original copies of the infected programs. Make working copies of these uninfected programs onto separate floppy diskettes, one sample program per diskette.
- Insert each floppy in turn into the infected system and run each sample program. This, in most cases, will cause the diskette, or the program, to become infected.
- Using a disk utility, do a binary compare of the infected diskette to the backup copy. If an infection has occurred, the diskettes will differ. Separate all working copy diskettes that have been modified by the virus and label them as infected.
- Now run the identification program against each of the infected floppies. Do this on a clean, uninfected system. The program should identify the infection on each diskette. Next cause the program to attempt removal. Run each floppy in turn through the removal cycle. The program should remove all of the infections.
- To test that the removal worked, take the infected (and now hopefully disinfected) diskettes and again do a binary compare against the original backup diskettes. There should be no discrepancy.
- If the program has passed the above tests, it is clearly able to identify and, at least in test disks, remove the infection. At this point you should test its operation on the infected system. To do this, first make a backup copy of the product. Then load the identification program into the infected system and begin the identification and disinfection process.

- On completion of the operation, perform a disk compare of the working disk against the original product disk. There should be no differences.

F. ANTI-VIRAL SOFTWARE

The following anti-viral products are available commercially:

1. Infection Prevention Systems

Product	Vendor	Phone
ACE System	Security Dynamics	(617) 547-7920
Data Physician	Digital Dispatch 1580 Rice Creek Road Minneapolis, MN 55432	(612) 571-7400
Disk Defender	Director Technologies	(312) 491-2334
Flu-shot+	Software Concepts Design 594 Third Avenue New York, NY 10016	(212) 889 6438
FShield	McAfee Associates 4423 Cheeney Street Santa Clara, CA 95054	(408) 988-3832
F-PROT	Box 7180 IS-127 Reykjavik Iceland	
Norton AntiVirus	10201 Torre Avenue Cupertino, CA 95014	(800) 343-4714
VIRALARM 2000	Lasertrieve 395 Main Street Metuchen, NJ 08840	(201) 906-1901
Virus Implant Protector	Leemah Datacom Security 3948 Trust Way Hayward, CA 94545	(415) 786-0790

2. Infection Detection Systems

Product	Vendor	Phone
Sentry	McAfee Associates	(408) 988 3832
Tracer	McAfee Associates	(408) 988-3832
Vaccinate	Sophco P.O. Box 7430 Boulder, CO 80306	(800) 922-3001
Virus-Pro	International Security Technologies	(212) 288-3101
Virus Safe	Enigma Logic Inc. 2151 Salvio Street, #301 Concord, CA 94565 USA	(415) 827-5707

3. Infection Identification Systems

Product	Vendor	Phone
Detect	McAfee Associates	(408) 988-3832
Scan Virus	IBM	(800) 426-2468
Viruscan	McAfee Associates	(408) 988-3832
V-Finder	WallyWare	(408) 275-6358

G. MD4 LISTING

```

/*
*****
** md4.c -- Implementation of MD4 Message Digest Algorithm          **
** Updated: 2/16/90 by Ronald L. Rivest                             **
** (C) 1990 RSA Data Security, Inc.                                  **
*****
*/
/*
** To use MD4:
** -- Include md4.h in your program
** -- Declare MDstruct MD to hold the state of the digest computation.
** -- Initialize MD using MDbegin(&MD)
** -- For each full block (64 bytes) X you wish to process, call
**     MDupdate(&MD,X,512)
**     (512 is the number of bits in a full block.)
** -- For the last block (less than 64 bytes) you wish to process,
**     MDupdate(&MD,X,n)
**     where n is the number of bits in the partial block. A partial
**     block terminates the computation, so every MD computation should
**     terminate by processing a partial block, even if it has n = 0.
** -- Message digest is available in MD.buffer[0] ... MD.buffer[3].
**     (Least-significant byte of each word should be output first.)
** -- You can print out the digest using MDprint(&MD)
*/

/* Implementation notes:
** This implementation assumes that ints are 32-bit quantities.
** If the machine stores the least-significant byte of an int in the
** least-addressed byte (VAX and 8086), then LOWBYTEFIRST should be
** set to TRUE. Otherwise (eg., SUNS), LOWBYTEFIRST should be set to
** FALSE. Note that on machines with LOWBYTEFIRST FALSE the routine
** MDupdate has a side-effect on its input array (the order of bytes
** in each word are reversed). If undesired, a MDreverse(X) call can
** reverse the bytes of X back into order after each call to MDupdate.
*/

#define TRUE 1
#define FALSE 0
#define LOWBYTEFIRST TRUE

/* Compile-time includes
*/

#include <stdio.h>
#include "md4.h"

/* Compile-time declarations of MD4 'magic constants'.
*/

#define I0 0x67452301 /* Initial values for MD buffer */
#define I1 0xefcdab89
#define I2 0x98badcfe
#define I3 0x10325476
#define C2 013240474631 /* round 2 constant = sqrt(2) in octal */
#define C3 015666365641 /* round 3 constant = sqrt(3) in octal */
/* C2 and C3 are from Knuth, The Art of Programming, Volume 2
** (Seminumerical Algorithms), Second Edition (1981), Addison-Wesley.
** Table 2, page 660.
*/
#define fs1 3 /* round 1 shift amounts */
#define fs2 7
#define fs3 11

```

```

#define fs4 19
#define gs1 3          /* round 2 shift amounts */
#define gs2 5
#define gs3 9
#define gs4 13
#define hsl 3          /* round 3 shift amounts */
#define hs2 9
#define hs3 11
#define hs4 15

/* Compile-time macro declarations for MD4.
** Note: The 'rot' operator uses the variable 'tmp'.
** It assumes tmp is declared as unsigned int, so that the >>
** operator will shift in zeros rather than extending the sign bit.
*/
#define f(X,Y,Z)      ((X&Y) | ((~X)&Z))
#define g(X,Y,Z)      ((X&Y) | (X&Z) | (Y&Z))
#define h(X,Y,Z)      (X^Y^Z)
#define rot(X,S)       (tmp=X,(tmp<<S) | (tmp>>(32-S)))
#define ff(A,B,C,D,i,s) A = rot((A + f(B,C,D) + X[i]),s)
#define gg(A,B,C,D,i,s) A = rot((A + g(B,C,D) + X[i] + C2),s)
#define hh(A,B,C,D,i,s) A = rot((A + h(B,C,D) + X[i] + C3),s)

/* MDprint(MDp)
** Print message digest buffer MDp as 32 hexadecimal digits.
** Order is low-order byte of buffer[0] to high-order byte of buffer[3].
** Each byte is printed with high-order hexadecimal digit first.
** This is a user-callable routine.
*/
void
MDprint(MDp)
MDptr MDp;
{ int i,j;
  for (i=0;i<4;i++)
    for (j=0;j<32;j=j+8)
      printf("%02x",(MDp->buffer[i]>>j) & 0xFF);
}

/* MDbegin(MDp)
** Initialize message digest buffer MDp.
** This is a user-callable routine.
*/
void
MDbegin(MDp)
MDptr MDp;
{ int i;
  MDp->buffer[0] = I0;
  MDp->buffer[1] = I1;
  MDp->buffer[2] = I2;
  MDp->buffer[3] = I3;
  for (i=0;i<8;i++) MDp->count[i] = 0;
  MDp->done = 0;
}

/* MDreverse(X)
** Reverse the byte-ordering of every int in X.
** Assumes X is an array of 16 ints.
** The macro revx reverses the byte-ordering of the next word of X.
*/
#define revx { t = (*X << 16) | (*X >> 16); \
  *X++ = ((t & 0xFF00FF00) >> 8) | ((t & 0x00FF00FF) << 8); }
MDreverse(X)
unsigned long int *X;
{ register unsigned long int t;
  revx; revx; revx; revx; revx; revx; revx; revx;
  revx; revx; revx; revx; revx; revx; revx; revx;
}

```

```

/* MDblock(MDp,X)
** Update message digest buffer MDp->buffer using 16-word data block X.
** Assumes all 16 words of X are full of data.
** Does not update MDp->count.
** This routine is not user-callable.
*/
static void
MDblock(MDp,X)
MDptr MDp;
unsigned long int *X;
{
    register unsigned long int tmp, A, B, C, D;
    #if LOWBYTEFIRST == FALSE
        MDreverse(X);
    #endif
    A = MDp->buffer[0];
    B = MDp->buffer[1];
    C = MDp->buffer[2];
    D = MDp->buffer[3];
    /* Update the message digest buffer */
    ff(A, B, C, D, 0, fs1); /* Round 1 */
    ff(D, A, B, C, 1, fs2);
    ff(C, D, A, B, 2, fs3);
    ff(B, C, D, A, 3, fs4);
    ff(A, B, C, D, 4, fs1);
    ff(D, A, B, C, 5, fs2);
    ff(C, D, A, B, 6, fs3);
    ff(B, C, D, A, 7, fs4);
    ff(A, B, C, D, 8, fs1);
    ff(D, A, B, C, 9, fs2);
    ff(C, D, A, B, 10, fs3);
    ff(B, C, D, A, 11, fs4);
    ff(A, B, C, D, 12, fs1);
    ff(D, A, B, C, 13, fs2);
    ff(C, D, A, B, 14, fs3);
    ff(B, C, D, A, 15, fs4);
    gg(A, B, C, D, 0, gs1); /* Round 2 */
    gg(D, A, B, C, 4, gs2);
    gg(C, D, A, B, 8, gs3);
    gg(B, C, D, A, 12, gs4);
    gg(A, B, C, D, 1, gs1);
    gg(D, A, B, C, 5, gs2);
    gg(C, D, A, B, 9, gs3);
    gg(B, C, D, A, 13, gs4);
    gg(A, B, C, D, 2, gs1);
    gg(D, A, B, C, 6, gs2);
    gg(C, D, A, B, 10, gs3);
    gg(B, C, D, A, 14, gs4);
    gg(A, B, C, D, 3, gs1);
    gg(D, A, B, C, 7, gs2);
    gg(C, D, A, B, 11, gs3);
    gg(B, C, D, A, 15, gs4);
    hh(A, B, C, D, 0, hs1); /* Round 3 */
    hh(D, A, B, C, 8, hs2);
    hh(C, D, A, B, 4, hs3);
    hh(B, C, D, A, 12, hs4);
    hh(A, B, C, D, 2, hs1);
    hh(D, A, B, C, 10, hs2);
    hh(C, D, A, B, 6, hs3);
    hh(B, C, D, A, 14, hs4);
    hh(A, B, C, D, 1, hs1);
    hh(D, A, B, C, 9, hs2);
    hh(C, D, A, B, 5, hs3);
    hh(B, C, D, A, 13, hs4);
    hh(A, B, C, D, 3, hs1);
    hh(D, A, B, C, 11, hs2);
    hh(C, D, A, B, 7, hs3);

```

```

    hh(B , C , D , A , 15 , hs4);
    MDp->buffer[0] += A;
    MDp->buffer[1] += B;
    MDp->buffer[2] += C;
    MDp->buffer[3] += D;
}

/* MDupdate(MDp,X,count)
** Input: MDp -- an MDptr
**        X -- a pointer to an array of unsigned characters.
**        count -- the number of bits of X to use.
**              (if not multiple of 9, uses high bits of last byte.)
** Update MDp using the number of bits of X given by count.
** This is the basic input routine for an MD4 user.
** The routine completes the MD computation when count < 512, so
** every MD computation should end with one call to MDupdate with a
** count less than 512. A call with count 0 will be ignored if the
** MD has already been terminated (done!=0), so an extra call with count
** 0 can be given as a 'courtesy close' to force termination if desired.
*/
void
MDupdate(MDp,X,count)
MDptr MDp;
unsigned char *X;
unsigned int count;
{ unsigned long int i, tmp, bit, byte, mask;
  unsigned char XX[64];
  unsigned char *p;
  /* return with no error if this is a courtesy close with count
  ** zero and MDp->done is true.
  */
  if (count == 0 && MDp->done) return;
  /* check to see if MD is already done and report error */
  if (MDp->done) { printf("\nError: MDupdate MD already done."); return; }
  /* Add count to MDp->count */
  tmp = count;
  p = MDp->count;
  while (tmp)
  { tmp += *p;
    *p++ = tmp;
    tmp = tmp >> 9;
  }
  /* Process data */
  if (count == 512)
  { /* Full block of data to handle */
    MDblock(MDp,(unsigned long int *)X);
  }
  else if (count > 512) /* Check for count too large */
  { printf
    (" \nError: MDupdate called with illegal count value %d.",count);
    return;
  }
  else /* partial block -- must be last block so finish up */
  { /* Find out how many bytes and residual bits there are */
    byte = count >> 3;
    bit = count & 7;
    /* Copy X into XX since we need to modify it */
    for (i=0;i<=byte;i++) XX[i] = X[i];
    for (i=byte+1;i<64;i++) XX[i] = 0;
    /* Add padding '1' bit and low-order zeros in last byte */
    mask = 1 << (7 - bit);
    XX[byte] = (XX[byte] & mask) & ~(mask - 1);
    /* If room for bit count, finish up with this block */
    if (byte <= 55)
    { for (i=0;i<8;i++) XX[56+i] = MDp->count[i];
      MDblock(MDp,(unsigned long int *)XX);
    }
    else /* need to do two blocks to finish up */

```

```

{ MDblock(MDp, (unsigned long int *)XX);
  for (i=0; i<56; i++) XX[i] = 0;
  for (i=0; i<8; i++) XX[56+i] = MDp->count[i];
  MDblock(MDp, (unsigned long int *)XX);
}
  /* Set flag saying we're done with MD computation */
  MDp->done = 1;
}
}

/*
** End of md4.c
*/

```

IX. LIST OF REFERENCES

Bosen, Bob, "Use of Message Authentication Code (MAC) Technology in the Detection of Computer Viruses", Enigma Logic, 1989, unpublished.

Chess, David, IBM, VIRUS-L Digest Tuesday, 8 May 1990 Volume 3 : Issue 90, moderated by Kenneth R. van Wyk.

Cohen, Fred, "Computer Viruses: Theory and Experiments", Computer Security: A Global Challenge, J.H. Finch and E.G. Dougall (eds), Elsevier Science Publishers, B.V. North Holland, 1984.

Davida, George, Desmedt, Yvo, and Matt, Brian, "Defending Systems Against Viruses Through Cryptographic Authentication", Proceedings 1989 IEEE Computer Society Symposium on Security and Privacy, IEEE Computer Society Press, Washington, DC, 1989.

Denning, Dorothy, "Cryptography and Data Security", Addison-Wesley Publishing Company, 1982.

McAfee, John, McAfee Associates, Viral Characteristics List V74, 1991.

McAfee, John, McAfee Associates, Implementing Anti-viral Programs, 1989.

McDonald, Chris, Information Systems Management Specialist, ASQNC-TWS-RA, VIRUS-L Digest Tuesday, 24 Apr 1990 Volume 3 : Issue 80, moderated by Kenneth R. van Wyk.

Merkle, Ralph, "A Fast Software One Way Hash Function", Xerox Corporation, 1989, unpublished.

Merkle, Ralph, "A Certified Digital Signature", Crypto 1989.

Mildner, John, Code 043, Naval Electronic Systems Security Engineering Center, Washington, DC, telephone conversation, 12 March 1991.

Rivest, Ron, RSA Data Security, February 1990, private communication.

Seberry, Jennifer and Pieprzyk, Josef, "Cryptography: An Introduction To Computer Security", Prentice-Hall, Australia, 1989.

Skulason, Fridrik, Technical Editor, Virus Bulletin (UK), University of Iceland, VIRUS-L Digest Friday, 9 Feb 1990 Volume 3 : Issue 35, moderated by Kenneth R. van Wyk.

Skulason, Fridrik, Technical Editor of the Virus Bulletin (UK), University of Iceland, VIRUS-L Digest Tuesday, 27 Mar 1990 Volume 3 : Issue 64, moderated by Kenneth R. van Wyk.

Skulason, Fridrik, Technical Editor of the Virus Bulletin (UK), University of Iceland, VIRUS-L Digest Wednesday, 9 May 1990 Volume 3 : Issue 91, moderated by Kenneth R. van Wyk.

Skulason, Fridrik, Technical Editor of the Virus Bulletin (UK), University of Iceland, as translated by Vesselin Bontchev, VIRUS-L Digest Friday, 18 May 1990 Volume 3 : Issue 97, moderated by Kenneth R. van Wyk.

Skulason, Fridrik, Technical Editor of the Virus Bulletin (UK), University of Iceland, VIRUS-L Digest Monday 7 May 1990 Volume 3 : Issue 89, moderated by Kenneth R. van Wyk.

Skulason, Fridrik, Technical Editor of the Virus Bulletin (UK), University of Iceland, VIRUS-L Digest Monday, 29 Jan 1990 Volume 3 : Issue 24, moderated by Kenneth R. van Wyk.

Spafford, Gene, Software Engineering Center, Purdue University, VIRUS-L Digest Friday, 12 Jan 1990 Volume 3 : Issue 10, moderated by Kenneth R. van Wyk.

Swimmer, Morton, Virus Test Center, University of Hamburg, VIRUS-L Digest Monday, 21 May 1990 Volume 3 : Issue 99, moderated by Kenneth R. van Wyk.

Tannenbaum, Andrew S., "Computer Networks", Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

INITIAL DISTRIBUTION LIST

- | | |
|--|---|
| 1. Defense Technical Information Center
Cameron Station
Alexandria, Virginia 22304-6145 | 2 |
| 2. Commanding Officer
Naval Electronic Systems Security Engineering Center
Code 043
3801 Nebraska Ave NW
Washington, DC 20393-5270 | 3 |
| 3. Dr. Norman F. Schneidewind
Code AS-SS
Naval Postgraduate School
Monterey, California 93943-5002 | 3 |
| 4. Dr. Tung X. Bui
Code AS-TN
Naval Postgraduate School
Monterey, California 93943-5002 | 1 |
| 5. Library, Code 52
Naval Postgraduate School
Monterey, California 93943-5002 | 2 |